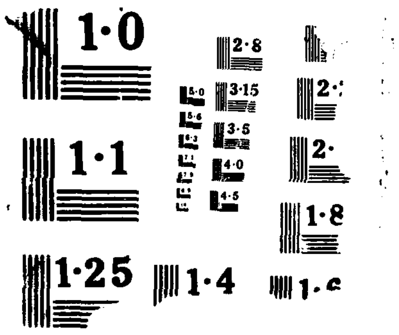
AD-A195 402   THE OPTIMIZATION OF AUTOMATICALLY GENERATED COMPILERS   1/2
(U) COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE
M L HALL 1987 N00014-86-K-0204

UNCLASSIFIED   F/G 12/5   NL


1·0
2·8
3·15
2·
5·0
5·6
3·6
1·1
4·0
2·
4·5
1·8
1·25
1·4

(4)

Contract N00014-86-K-0204

The Optimization of Automatically Generated Compilers

by

Mark Lee Hall

B.S. College of William and Mary in Virginia, 1982

A *thesis* submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

1987

This thesis for the Doctor of Philosophy degree by

Mark Lee Hall

has been approved for the

Department of Computer Science

by

[illegible]

William M. Waite

[illegible]

Lloyd D. Fosdick

[illegible]

Harry F. Jordan

[illegible]

Carlos A. Felippa

[illegible]

Vincent P. Heuring

[illegible]

Clayton H. Lewis

Date April 22, 1987

Hall, Mark Lee (Ph.D., Computer Science)

The Optimization of Automatically Generated Compilers

Thesis directed by Professor William M. Waite

An "Attribute Grammar" (AG) is a formalism for specifying the "semantics" of a computer language. Specifically, an AG specifies the rules for creating a "structure tree" for a string, and evaluating the "attributes" of the tree. The result of this "attribute evaluation" (AE) is a "decorated" tree representing the semantic properties of the string.

From a software engineering standpoint, AGs have proven to be very useful in the area of compiler specification and automatic construction. This is due chiefly to three outstanding features: (1) AGs are "declarative" specifications, which are generally accepted to be easier to write and less error prone than their procedural counterparts, and are also easier to analyze for storage optimizations; (2) AGs can be algorithmically checked to be "non-circular" and "complete," which guarantees that all attributes in a structure tree are computable; and (3) AEs can be mechanically derived from AG specifications.

These benefits are unavailable to those who write AEs for compilers "by hand".

Unfortunately, compilers which are automatically generated from AGs often require tremendous amounts of storage. This thesis focuses on the reduction of this storage problem. ~~by:~~ Keywords: theses; algorithms; parsing.

(1) Providing algorithms to move the storage for many attributes from the structure tree into global stacks and variables.

(2) Creating AEs which build and evaluate structure tree "fragments" during parse time, in an attempt to keep instantaneous storage requirements to a minimum.

DEDICATION

I dedicate this,

the culmination of twenty-two years

of formal education,

to Ruby and Ralph, Shirley and Richard,

and especially

Tamara and Sparky

ACKNOWLEDGEMENTS

The primary acknowledgement for this thesis must go to Prof. William Waite, who wanted me to get a Ph.D. and get out into the world even more than I did.

Special thanks goes to Dotty, who knows what every administrative rule, regulation, and deadline is to the last detail. More special thanks goes to Evi, who kept after me until I actually started looking for jobs.

Thanks goes to Professor Hal Gabow, who bailed me out when I was trying to transform the attribute partitioning problem to the Maximum Network Flow problem in Chapter III of this thesis. When things really get sticky, you should never be too far from a good theoretician.

Thanks goes to Tamara, who gave me the inspiration and support to do Ph.D. research by marrying me. I also thank my other comrades of the past five years: Paul, who lives right; Mark, who knows everything about troff; Bob, who gives help for free; Jim and Tony, for "48 hours"; and Mary Jo and Bonnie, for the belly dancer.

The research for this thesis was generously funded by grants from the Army Research Office: DAAL 03-86-K-0100, and from the Office of Naval Research: N00014-86-K-0204.

CONTENTS

CHAPTER I

INTRODUCTION TO ATTRIBUTE GRAMMARS AND COMPILER GENERATION

1.1. Advantage of formal compiler specification

It is generally accepted that large scale software engineering efforts are well aided by the combination of sound methodologies and software tools. The creation of compilers for programming languages, being a large scale yet well understood software undertaking, has long been a model for the study of such methodologies and tools. From this study, a particularly sound combination, the use of attribute grammars (AGs) [Knu68] and compiler generator systems, has emerged which shows great promise. AGs are a natural formalism for specifying the computations required to carry out the correct translation of a computer language. They have been manageably used to specify sophisticated compilers for languages such as Ada and Pearl [KHZ82], and are a method of choice for language designers who are also interested in implementation issues.

Tool support for this area is extensive; programs exist which can automatically verify the "completeness," "consistency," and "non-circularity" of an AG, important properties which raise the confidence that a compiler will not fail in practice. Other programs exist which can mechanically instantiate a working compiler from an AG.

Apart from the mechanical utility the tools provide when using AGs, they also provide the ability for compiler writers to debug at the **specification** level. At this level, the sources of bugs in the compiler are more likely to be located, and fixes are more likely to produce intended behavior.[1] It is this advantage over hand coding that points to the real contribution that AGs can give to the future of compiler writing (and of software engineering in general): the ability to program ''in-the-large'' as described in [DeK75].

1.2. The problem with attribute grammars

Given the ease and confidence with which compilers can be produced using AGs, it is surprising that they are not more widely used. The reason for this appears to be the same problem that other programs generated from specifications have; they tend to be much larger and slower than their hand written counterparts. Historically, both automatically generated lexical analyzers and automatically generated parsers have suffered from sluggish performance [Wai86b], [Wai85a]. Similarly, compilers generated by existing AG based systems often suffer from gigantic storage and runtime requirements. Many people feel that the performance is so unacceptable that they will revert to hand coding just to get better results [ScJ83]. Others find the completeness information useful, and write AGs for languages for this reason only [Wai85b]. Thereafter, the specification is coded up by hand. This allows the incorporation of very fast semantic evaluation routines, which can speed performance markedly. It also allows attribute analysis to be integrated with the parsing phase, which enhances both speed and space characteristics (an explicit interface between parsing and semantic evaluation is no longer required to be built and separately traversed).

Unfortunately, once a hand written compiler exists, whatever specifications were used to describe its intended properties are largely ignored,

[1]As an example, consider correcting an infinite loop in a hand written parser. Was it caused by a programmer error, or was it caused by an indirectly left recursive grammar? Non-trivial debugging may be required to know for sure. Certainly it is much easier to have a parser generator indicate that the grammar is left recursive, and have the grammar writer correct the error.

meaning that the addition of new language features and debugging must be carried out using the hand coded program as a reference. This is analogous to adding new grammar features and debugging the source code **output** of a parser generator when the **grammar** needs modifications! While this would never be done for automatically generated parsers, the unfortunate reality of automatically generated compilers is that often the overwhelmingly simpler method of fixing the error at the specification level is forsaken in lieu of the significantly better performance which can be squeezed out by hand coding.

In order to remedy the avoidance of AG use among compiler writers, this thesis will expose the major areas where automatically generated compilers fail to perform optimally, and will show how these shortcomings may be fixed. The result of this will hopefully bring automatically generated compilers much closer to the speeds and storage requirements of hand written compilers, which should allow more people to enjoy the benefits of *programming by specification*.

1.3. Attribute grammar background

A context-free grammar (CFG) [HoU79] consists of a group of ''grammar symbols'' (or just ''symbols''), each of which represents a group of strings. These symbols are defined using primitive ''terminal'' symbols of the grammar, or recursively in terms of the remaining ''non-terminal'' symbols. Rules for stating the direct and recursive associations between symbols are known as ''productions.'' Formally, a CFG is denoted by $G=(N,T,P,Z)$ where N is the set of non-terminals, T is the set of terminals, P is the set of productions, and Z is the ''start'' symbol, which cannot appear on the right hand side of any production in P. The set $V=N \cup T$ is conveniently referred to as the ''vocabulary'' set for G. Each production $p \in P$ has the form

$$p: X ::= \alpha$$

where $X \in V$ and $\alpha \in V^*$. The relation => is defined over strings in V^* as follows: if $p: X ::= \alpha$ is a production of P and $\beta X \gamma, \beta\alpha\gamma \in V^*$, then $\beta X \gamma => \beta\alpha\gamma$ ($\beta X \gamma$ "directly derives" $\beta\alpha\gamma$ via production p). The relation $=>^*$ is the transitive closure of =>. A string w is said to be a member of the "language" $L(G)$ iff $Z =>^* w$ exists using the productions of G. The process for obtaining a derivation for w is known as its "parse."

It is often convenient to represent the sequence of productions used to derive $w \in L(G)$ by a "parse tree." The "nodes" of such a tree correspond to non-terminals of the grammar, and the "leaves" correspond to terminals. Suppose $X_i \in V$ and

$$p: X_0 ::= X_1 X_2 \cdots X_n$$

is a production of P[2]. If p is used in the derivation of w, then the parse tree for w contains at least one "production instance" of p, where X_0 is the "parent" node of the instance and each $X_i, i>0$ is a "child" node. The child nodes are either leaves of the tree or are themselves parents of some other production instance in the tree. Figure 1.1a gives an example of a CFG taken from [Knu68] (and simplified in [JaP81]) which describes the syntax of binary numbers. As an example, the parse of the binary number 101.011 yields the parse tree of Figure 1.1b. For clarity, the nodes of the parse tree have been labeled with the productions (and their left hand side non-terminal symbols) used to derive w. The leaves have been labeled with their corresponding terminals.

In its present form, a CFG G embodies nothing more than the context "free" properties of $L(G)$. In order to specify context "sensitive" (or

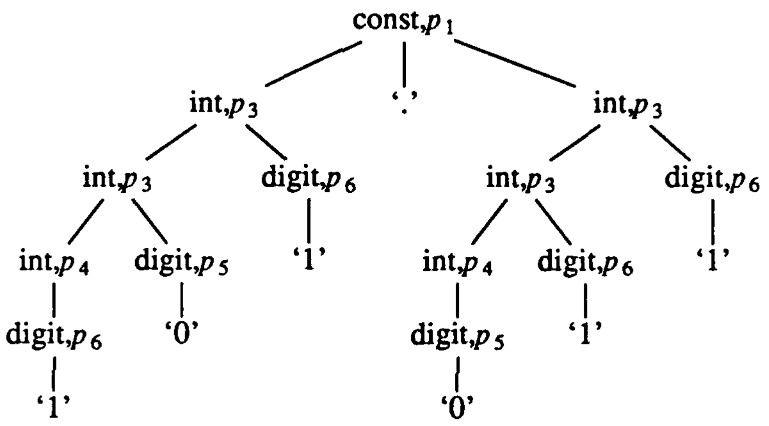
[2]In this thesis, if the same symbol X is used twice in a production, the two occurrences are distinguished by the symbols $X[i], X[j]$. The symbols X_i, X_j in general denote the symbols in the ith and jth locations of a production, with the left hand side symbol being X_0.

Symbols $V(G)$={const, int, digit, '.', '0', '1'}

Productions $P(G)$=

p_1 : const ::= int '.' int
p_2 : const ::= int
p_3 : int ::= int digit
p_4 : int ::= digit
p_5 : digit ::= '0'
p_6 : digit ::= '1'

(a) - binary number context free grammar



(b) - parse tree for binary number 101.011

- Figure 1.1: Binary Number Context Free Grammar and Sample Parse Tree -

''semantic'') properties as well, G can be augmented by ''attribute computations'', yielding an ''attribute grammar'' (AG) $\bar{G}$. In $\bar{G}$, a set of ''attributes'' is associated to each original symbol of G, and semantic ''rules'' are added to each production of G which specify the interaction between the attributes appearing in that production. The set of attributes for a symbol X is denoted $A(X)$. Occurrences of $a \in A(X)$ in $\bar{G}$ are denoted by $X.a$, to distinguish them from $a \in A(Y)$ for symbols $Y \neq X$. The j^{th} semantic rule for a production $p_i \in P$ has the form

$$r_{i,j} : X.a := f(\cdots Y.b \cdots)$$

where X, Y occur in p_i, and $a \in A(X), b \in A(Y)$. $X.a$ is said to be the ''target'' of $r_{i,j}$, and can be the target of at most one rule in any one production. As with parse trees for G, "structure" trees can be created whose nodes are connected according to G, and "evaluated" according to $\overline{G}$. A completely evaluated structure tree represents the syntax **and** semantics of some string of $L(G)$. For example, Figure 1.2 shows an AG (based on the CFG of Figure 1.1) which computes the value of any syntactically correct binary number. Intuitively, the attributes *scale* and *length* provide information about the context in which each *digit* occurs, so that the contribution a particular *digit* makes to the overall *value* of a string can be determined.

Symbols and Attributes

$V(G)$	$A(X)$
const	{value}
int	{scale, value, length}
digit	{scale, value}

Productions and Rules

p_1 : const ::= int '.' int
- $r_{1,1}$: const.value := int[1].value + int[2].value;
- $r_{1,2}$: int[1].scale := 0;
- $r_{1,3}$: int[2].scale := – int[2].length;

p_2 : const ::= int
- $r_{2,1}$: const.value := int.value;
- $r_{2,2}$: int.scale := 0;

p_3 : int ::= int digit
- $r_{3,1}$: int[1].value := int[2].value + digit.value;
- $r_{3,2}$: int[1].length := int[2].length + 1;
- $r_{3,3}$: int[2].scale := int[1].scale + 1;
- $r_{3,4}$: digit.scale := int[1].scale;

p_4 : int ::= digit
- $r_{4,1}$: int.value := digit.value;
- $r_{4,2}$: int.length := 1;
- $r_{4,3}$: digit.scale := int.scale;

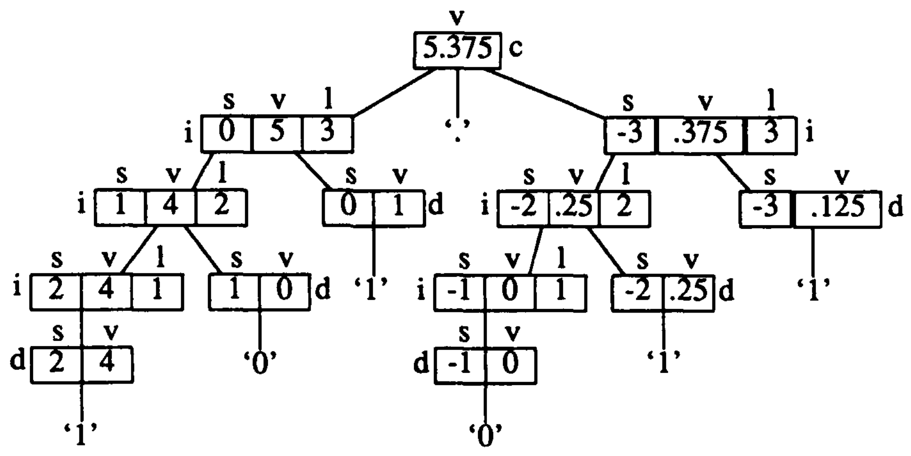
p_5 : digit ::= '0'
- $r_{5,1}$: digit.value := 0;

p_6 : digit ::= '1'
- $r_{6,1}$: digit.value := $2^{\text{digit.scale}}$;

- Figure 1.2: Binary Number Attribute Grammar -

Figure 1.3 shows the values of all the attributes required in the computation of the semantics of the binary number 101.011. The "distinguished" attribute *const.value* of the root node contains the final meaning of the string: the decimal value 5.375.

For the present discussion, it suffices to describe a parallel method of attribute evaluation that assigns a processor to each production instance in the tree. Each processor waits until all the arguments are "available" (have been computed) for one of the semantic rules associated to its production. It then evaluates that rule, and signals to other processors that the attribute indicated as the target of the rule is now available. This general method of attribute computation suffices to evaluate the attributes for any structure tree constructed from the class of "well-defined" AGs as in [Knu71].



c - "const" node
i - "int" node
d - "digit" node
v - "value" attribute
l - "length" attribute
s - "scale" attribute

- Figure 1.3: Evaluated Structure Tree for Binary Number 101.011 -

1.4. Attribute evaluator implementation and optimization

Using the parallel model, the runtime for evaluating a structure tree T is never any worse than $O(|A(T)|)$ where $|A(T)|$ is the number of attribute instances in T. When implemented on a sequential computer, however, the overhead involved in modeling the parallel evaluator is prohibitively expensive. Instead, a more disciplined approach must be taken to ensure that the evaluator will not have to search for attributes to evaluate.

1.4.1. Tree walk automata

For the sequential machine, it is best if the evaluator simply "knows" which attributes in a production instance are available ahead of time, so that extensive bookkeeping is not required. For example, consider the structure tree of Figure 1.4, which contains the example structure tree of the binary number 101.011, such that each production instance has been augmented with a "visit sequence." Briefly, a visit sequence is a left to right, read only sequential input tape which is consumed by a finite state machine known as a "tree walk automata" (TWA) [Kam83]. The operations a TWA makes as it processes a structure tree are:

(1) (*EVAL*) if the TWA is currently "visiting" a tree node labeled X_0,p_i, where p_i is the production

$p_i : X_0 ::= X_1 \cdots X_n$

with semantic routine

$r_{i,j} : X_k.a := f(\cdots)$

and the current action on the input tape at this tree node is

$\rightarrow X_k.a$

then the TWA consumes the input action, and computes the expression on the right hand side of $r_{i,j}$, storing the result into the attribute instance $X_k.a$.

(2) (*VISIT*) if the TWA is currently visiting a tree node labeled X_0,p, where p is the production

$p : X_0 ::= X_1 \cdots X_n$

and the current action on the input tape at this tree node is

$\downarrow_j X_i$ where $i,j>0$

then the TWA consumes the input action and moves "down" to the child node X_i,q for the j^{th} time (where q is some production having X_i as left hand symbol). This action is historically known as a "visit."

(3) (*LEAVE*) if the TWA is currently visiting a tree node labeled X_0,p, where p is the production

$p : X_0 ::= X_1 \cdots X_n$

and the current action on the input tape at this tree node is

$\uparrow_j$ where $j>0$

then the TWA consumes the input action and moves "up" to the parent of node X_0,p for the j^{th} time. This action is historically known as a "leave."

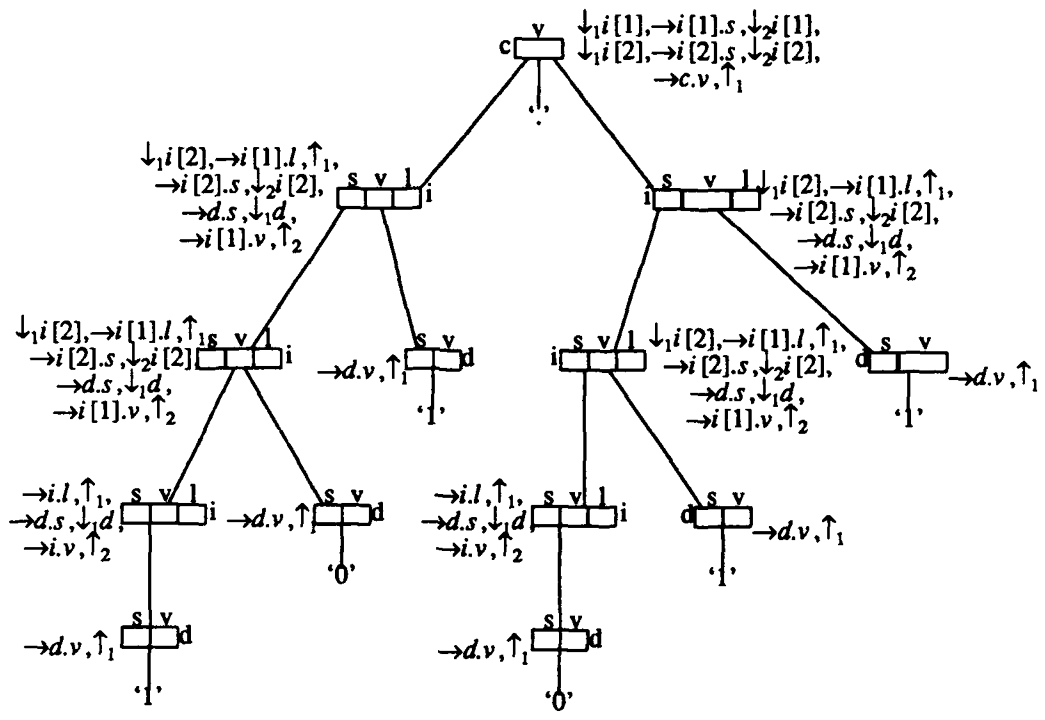
For the structure tree of Figure 1.4, the execution of TWA begins with an initial $\downarrow_1 const$, and ends when the TWA consumes the final action $\uparrow_1$ on the input tape at the node $const\_p_1$ in the structure tree. The execution of TWA over this structure tree can be easily shown to produce the same results as Figure 1.3 (the reader unfamiliar with TWAs is encouraged to execute several steps for the tree in this figure).

Given any attribute grammar G which belongs to the class of simple multi-visit AGs [EnF82], visit sequences are guaranteed to exist for the productions of G such that a TWA using them will successfully evaluate all attributes of any possible structure tree for G.

1.4.2. Global stacks and variables

Now that a method for attribute evaluation with sufficiently good runtime has been described, the next step is to bring the storage requirements down to a reasonable level. In a naive implementation of the structure tree, all attributes for a node are declared as in a Pascal record, having one named field for each attribute. During the parse, the structure tree nodes are dynamically allocated and strung together according to the productions used. Unfortunately, even small input strings typically produce large structure trees [Fan72], so this method is unmanageable in practice. For example, in [Gal81], it is reported that some 10 million bytes of attribute storage is required to analyze a 50-100 line Ada program using this technique!

The usual method for reducing the space requirements for a TWA is to remove certain attributes from the structure tree, where possible, and store all their instances in global variables and global stacks. Once those attributes are discovered which can be stored globally, the visit sequences can be changed to include assignment of values into globally declared variables and stacks (and also



c - "const" node v - "value" attribute
i - "int" node l - "length" attribute
d - "digit" node s - "scale" attribute

- Figure 1.4: Visit Sequences for Binary Number Structure Tree -

to include stack operations where required). For example, straightforward analysis of the visit sequences in Figure 1.4 shows that (at least) the attributes *digit.scale*, *digit.value*, and *int.length* can be implemented using global variables, and *int.scale* can be implemented as a global stack. The visit sequences of Figure 1.5 are the same as those of Figure 1.4 except for use of globally declared variables *digit_scale*, *digit_value*, *int_length*, and *int_scale* (and corresponding stack operations). The execution of a TWA given these augmented visit sequences can still be easily shown to produce the correct value 5.375 for the distinguished attribute *const.value* (as was produced in Figure 1.3). The only difference is that the structure tree will not contain storage for any instances of the attributes *digit.scale*, *digit.value*, *int.length* or *int.scale*. This significantly reduces the attribute storage required to evaluate even this small example.

The removal of attributes from the structure tree has proven to be a remarkably effective optimization. In [KHZ82], they report that some 85% of all attributes can be stored globally for an Ada attribute grammar. This is an impressive figure, but the actual implementation of storage optimizations, as excellently pointed out in [FaY86], is still less than optimal. Such unoptimality is significant; storing only 15% of Ada attributes in a structure tree might lower the storage requirements from 10 million bytes to 1.5 million bytes, but the number is still too high. Further research is clearly needed to reduce this number to something more manageable. In Chapter II of this thesis, results will be shown that, given a set of visit sequences to optimize over, the number of attributes stored in the structure tree can be decreased considerably. Chapter III will provide methods for the further reduction of tree based attributes by showing how to create visit sequences that are more conducive to optimization.

Global Variables:

digit_scale, digit_value, int_length: integer;
int_scale: **Stack Of** integer;

Visit Sequences:

/*appearance of rule $r_{i,j}$ (having target $X.a$) in $\rightarrow$*TOP*() action indicates that the semantic rule $r_{i,j}$ should be evaluated to get the proper value; attribute instance $X.a$ is on top of stack. /*global_var*/ indicates assignment to a global variable.
*/

p_1: $\downarrow_1$int[1], *PUSH*(int_scale), $\rightarrow$*TOP*(int_scale,$r_{1,2}$),
$\downarrow_2$int[1], *POP*(int_scale), $\downarrow_1$int[2],
PUSH(int_scale), $\rightarrow$*TOP*(int_scale,$r_{1,3}$),
$\downarrow_2$int[2], *POP*(int_scale), $\rightarrow$const.value, $\uparrow_1$

p_2: $\downarrow_1$int, *PUSH*(int_scale), $\rightarrow$*TOP*(int_scale,$r_{2,2}$),
$\downarrow_2$int, *POP*(int_scale), $\rightarrow$const.value, $\uparrow_1$

p_3: $\downarrow_1$int[2], $\rightarrow$int_length /*global_var*/, $\uparrow_1$,
PUSH(int_scale), $\rightarrow$*TOP*(int_scale,$r_{3,3}$),
$\downarrow_2$int[2], *POP*(int_scale), $\rightarrow$digit_scale /*global_var*/,
$\downarrow_1$digit, $\rightarrow$int[1].value, $\uparrow_2$

p_4: $\rightarrow$int_length /*global_var*/, $\uparrow_1$, $\rightarrow$digit_scale,
$\downarrow_1$digit, $\rightarrow$int.value, $\uparrow_2$

p_5: $\rightarrow$digit_value, $\uparrow_1$

p_6: $\rightarrow$digit_value, $\uparrow_1$

Visit Sequences of Figure 1.4 with Global Variables,
Global Stacks, and Stack Operations

- Figure 1.5 -

Currently, some well optimized high level language compilers can produce code rivaling that of expert assembly language programmers. Along those lines, future compiler generators might, with the inclusion of the optimizations given in this thesis, produce attribute evaluators whose storage requirements are just as good (or possibly better) than those of hand written evaluators.

1.4.3. Parse time attribution

Although justifiably satisfied with the enormous space optimizations over the attributes in their Ada and Pascal AGs, [KHZ82] nevertheless point out that using the structure tree as an explicit interface between the parsing and semantics phases of compilation increases space requirements dramatically. In addition, hand written single-pass compilers that perform semantic evaluation during parsing have been shown to translate programs more than four times faster than their TWA-based counterparts [Kos84]. While this figure certainly points out a need to integrate attribute evaluation with parsing, it is somewhat misleading. When measuring compilers for performance, one must account for the quality of the object code produced. In general, high quality code generation strictly demands the use of some intermediate representation of the input program, be it a structure tree or some other form.[3] Just because a hand written compiler runs fast does not mean that it is the most desirable one to use. Sometimes it only means that the compiler does not generate very good code. This has important ramifications for compiler generation. When good code must be produced by a hand written compiler, the implementor usually follows historical precedent (instead of specific

[3]For example, given an expression involving multiplication on a PDP11, to get the least register usage, one must sometimes decide to emit code for the right operand before the emitting the code to evaluate the left operand [WaG83,p. 260]. Of course, to make this decision, some internal representation of the expression will have to be evaluated; also the representation of the left operand will have to be stored while code is generated for the right subtree.

requirements) to decide where to build structure trees during compilation. This can lead to building structure trees unnecessarily. For example, the majority of recent C compilers are based on the Portable C Compiler as described in [Joh80]. This compiler builds structure trees for variable and type declarations and for expressions. There has never been any investigation as to whether or not structure trees are actually **necessary** for these constructs. Furthermore, we observe the following properties of this hand written system:

- No properties concerning the completeness or consistency of the evaluator actions is automatically available.
- No optimization of the structure tree or evaluation mechanism is automatically provided.

It should be pointed out that processing declarations for many block structured languages can be done during the parse phase without building structure trees. Using AGs, the exact set of structure trees which are **required** to be built for certain grammar constructs can be discovered, and automatic structure tree creation and attribute analysis can optimize these trees better than one would expect for a typical hand written compiler. In the case of C, a generated C compiler might thereby process declarations faster, using less space, than the portable C compilers.

Chapter IV of this thesis will deal with the integration of semantic analysis with parsing in a generated compiler. In contrast to the current hand written approach, this will provide the following functionality for AG writers:

- It will enable them to generate compilers with no structure tree where that is possible.
- It will enable them to have the exact structure tree built in those instances where one is needed.
- It will allow them to reduce the structure trees required by experimentally changing the **specification** of the compiler.

This should enable high quality compilers to be built with performance characteristics similar to their hand written counterparts, but with greatly reduced implementation effort.

CHAPTER II

ATTRIBUTE STORAGE OPTIMIZATIONS FOR FIXED VISIT SEQUENCES

2.1. Simple multi-visit attribute grammars

A particularly advantageous aspect of the simple multi-visit AGs is that the visit sequences for a production do not depend on the context in which the production appears in a structure tree. For example, reinspection of Figure 1.4 shows that the visit sequences are the same for each production instance in the structure tree. However, the visit sequences for the instances of production p_3 in the left subtree could be much different than those for p_3 in the right subtree. Specifically, the visit sequences for p_3 in the left subtree might be:

$$\rightarrow \mathit{int}[2].\mathit{scale}, \downarrow_1 \mathit{int}[1], \rightarrow \mathbf{\mathit{digit.scale}}, \downarrow_1 \mathit{digit}, \rightarrow \mathit{int}[1].\mathit{value}, \rightarrow \mathit{int}[1].\mathit{length}, \uparrow_1$$

Although this is a "simpler" visit sequence in some respects (there is only one *leave* from the production), it nonetheless requires the additional computation (at TWA runtime) of context information to determine that this visit sequence can actually be used. Moreover, the dynamic nature of this decision means that the recognition of some storage optimizations must also be moved to TWA runtime. These are both undesirable characteristics. Simple multi-visit AGs do not suffer from this: all visit sequence selection and storage optimization can be performed "statically" (at compiler generation time). Furthermore, simple multi-visit AGs have proven to be both sufficient and natural in the description of actual large scale programming languages [KHZ82]. Hence they are the grammars of choice for space and time-efficient attribute evaluation.

The idea which forms the basis for the class of simple multi-visit AGs is, of course, the notion of "multi-visit." A symbol X is multi-visit if it is the parent symbol of a production p where the visit sequence for p contains more than one *LEAVE* to X's parent symbol. Conversely, if X is the parent symbol of no production containing multiple *LEAVE*s then it is a "single-visit" symbol. For example, the symbol *int* of Figure 1.4 is multi-visit, while *const*,*digit* are single visit symbols.

The single visit property can be similarly defined for attributes, but with a subtle difference. The "lifetime" of an instance of attribute $X.a$ is defined as being the sequence of actions executed by the TWA between the action $\rightarrow X.a$ which evaluates the instance and the last action $\rightarrow Y.b$ which uses the instance. As it stands, this definition of lifetime is too dependent on the TWA runtime to be useful for storage analysis. However, we can give an approximation to this by statically analyzing visit sequences. Suppose the symbol X is a child symbol of a production p. At some point in the visit sequence for p, $X.a$ is evaluated (either at $\rightarrow X.a$ or $\downarrow_i X$). At some later point in the visit sequence for p, $X.a$ is used for the last time (either at $\rightarrow Y.b$ or $\downarrow_j X$). We refer to the sequence of actions between these two points in p to be the "textual" lifetime of $X.a$ in p. If, during the textual lifetime of $X.a$ in p, no $\uparrow_k$ occurs, then $X.a$ is said to be a "single visit" attribute for p. If $X.a$ is a single visit attribute for all productions where X is a child symbol, then $X.a$ is "single visit." Conversely, if a just one production exists which violates the single visit property, then $X.a$ is "multi-visit."

It is important to point out that the single visit property for $X.a$ is not decided by what happens when X is the **parent** symbol of a production. For example, given the following AG fragment:

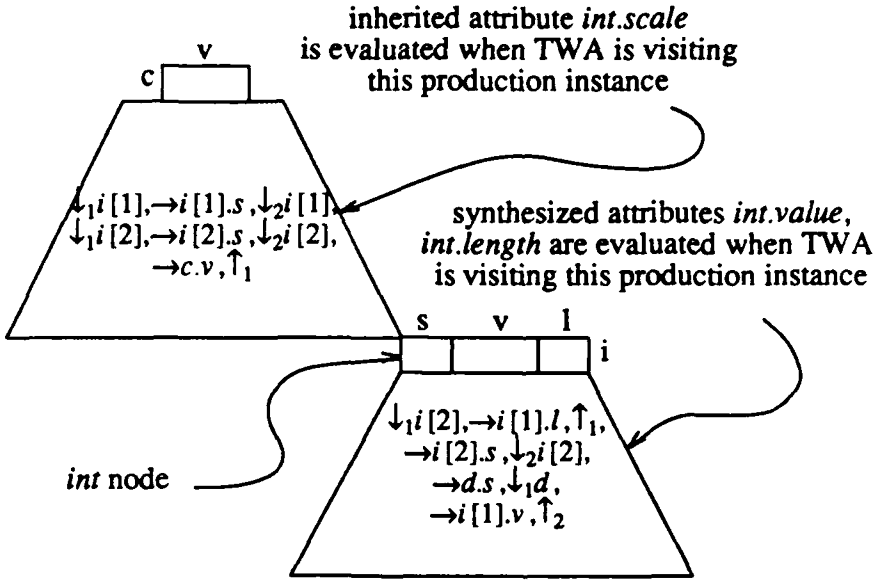
$p : X ::= Y\ Z$
  $r_{p,j} : Z.c := f(\cdots X.a \cdots);$

having visit sequence:

$$\cdots, \rightarrow X.a, \cdots, \uparrow_1, \cdots, \rightarrow Z.c, \cdots$$

the fact that the lifetime of $X.a$ spans a *LEAVE* in this production does **not** mean that $X.a$ is multi-visit. This distinction, namely defining single visit attributes in terms of **child** symbols, allows more attributes to be single visit than historically permitted (as in [FaY86, Saa78]). This, in turn, allows for more storage optimizations.

The single/multi-visit property is an important criterion for implementing attributes as global variables and stacks. Consider what happens when a visit sequence for an instance of production p in a structure tree calls for a *LEAVE*. When the TWA executes this action, it can visit virtually every other node in the structure tree before returning to that instance of p. Subsequently, if an attribute $X.a$ in p is multi-visit, then the probability is high that when the TWA *LEAVE*s in the middle of $X.a$'s lifetime, some unknown amount of other $X.a$ instances will be evaluated before the TWA returns. Such an $X.a$ cannot be statically implemented as a global stack since, upon return, the other $X.a$ instances might "bury" the original $X.a$ at some unknown stack depth. On the other hand, since the TWA performs no *LEAVE* during the lifetime of a single visit attribute, if the attribute is stacked it will not be "buried" when the TWA returns from "below." In addition, if a production has multiple instances of the same symbol, such that multiple instances of the same attribute can be alive at the same time, we can still use stacks since it is determinable at compiler generation time which of the instances will be at which stack location at run time (this will be proven over the remainder of this chapter).


inherited attribute *int.scale* is evaluated when TWA is visiting this production instance
v
c
$\downarrow_1 i[1], \rightarrow i[1].s, \downarrow_2 i[1],$
$\downarrow_1 i[2], \rightarrow i[2].s, \downarrow_2 i[2],$
$\rightarrow c.v, \uparrow_1$
synthesized attributes *int.value*, *int.length* are evaluated when TWA is visiting this production instance
s
v
l
i
int node
$\downarrow_1 i[2], \rightarrow i[1].l, \uparrow_1,$
$\rightarrow i[2].s, \downarrow_2 i[2],$
$\rightarrow d.s, \downarrow_1 d,$
$\rightarrow i[1].v, \uparrow_2$

Inherited Attributes Evaluated in "upper" Context
Synthesized Attributes Evaluated in "lower" Context
(a)

$V(G)$	$AI(X)$	$AS(X)$
const	{}	{value}
int	{scale}	{value, length}
digit	{scale}	{value}
'.'	{}	{}
'1'	{}	{}
'0'	{}	{}

Inherited and Synthesized Attributes
of Figure 1.2
(b)

- Figure 2.1 -

2.2. Synthesized and inherited attributes

When reasoning about the lifetimes of attributes and how to store them, it is useful to categorize each attribute according to the set of visit sequences containing its evaluation action. Consider the visit sequence for the production

const ::= *int* '.' *int*

in Figure 1.4. As shown in Figure 2.1a, a particular instance of *int.scale* is always computed when the tree walk evaluator is visiting the **parent** of the *int* node in the structure tree. That is because the evaluation action for *int.scale* always occurs in an "upper production context" of which *int* is a child symbol. On the other hand, instances of *int.value* and *int.length* are evaluated when the tree walk evaluator is visiting the *int* node itself. That is because the evaluation actions for *int.value* and *int.length* always occur in a "lower production context" of which *int* is the parent symbol. Attributes like *int.scale* are termed "inherited" (they "inherit" values from their parent nodes). Attributes like *int.value* and *int.length* are termed "synthesized" (they are "synthesized" from their subparts). The set of inherited attributes of a symbol X is denoted $AI(X)$, while the set of its synthesized attributes is denoted $AS(X)$. As an example, Figure 2.1b contains the sets of inherited and synthesized attributes for all the symbols of the example AG in Figure 1.2.

2.3. Implementing attributes using global variables

The best way in which to implement both synthesized and inherited attributes is obviously to do so using global variables. This avoids both the huge space requirement of storing tree-based attributes and the additional runtime requirement of pushing and popping stack-based attributes. Unfortunately, an attribute's lifetime must satisfy a very restrictive property for the attribute to be implemented as a global variable. This property is expressed in terms of the runtime of the TWA as follows: if **any** two instances of an attribute $X.a$ are "alive" (the TWA has defined them both but has not yet reached the last use of either), then $X.a$ cannot be implemented as a global variable.

For simple multi-visit AGs, the overlap of single visit attributes can be determined precisely by inspection of the visit sequences: given an attribute grammar G we add the action $\uparrow_0$ to the beginning of every visit sequence. Then, for production p containing symbol X, the "birth" of an attribute $X.a$ (as far as p is concerned) is one of the following actions:

(1) $\rightarrow X.a$ where $a \in AS(X)$ and X is the parent symbol of p, or

(2) $\downarrow_i X$ where $a \in AS(X)$ and X is a child symbol of p, or

(3) $\rightarrow X.a$ where $a \in AI(X)$ and X is a child symbol of p, or

(4) $\uparrow_i$ where $a \in AI(X)$ and X is the parent symbol of p.

Likewise, the "death" of an attribute $X.a$ (as far as p is concerned) is one of the following actions:

(1) $\downarrow_i X$ or $\rightarrow Y.b$ if X is a child symbol of p, or

(2) $\uparrow_i$ or $\rightarrow Y.b$ if X is the parent symbol of p.

As described before, these textual delimiters of an attribute's lifetime make it possible to decide which attributes can be implemented as global variables by context-free inspection of each individual production (ie, without having to consider combinations of productions). The algorithm of Figure 2.2 finds attributes which can be implemented as global variables based on the idea of textual lifetime. Each entry of the matrix $CAN\_EVAL[S,v]$ is a set containing those attributes which can possibly be evaluated during visit v to symbol S, including subvisits (*CAN_EVAL* is a variant of the *CAN_REFERENCE* information proposed by [FaY86]). This matrix allows the algorithm to find the **exact** set of single visit

```
/* On Entry: X.a∈SINGLE_VISIT iff X.a is single visit.
           X.a∈CAN_EVAL[S,v] IFF X.a can be evaluated during
                 visit v to symbol S.
   On Exit: GLOBAL_VAR[X.a]=True IFF X.a can be implemented as a global
                 variable */

/*find those attributes with overlapping lifetimes*/
foreach X.a∈SINGLE_VISIT do
   GLOBAL_VAR[X.a] := TRUE; /*assume it is a global var, then prove otherwise*/
   foreach production p containing X do
      foreach occurrence X[i] in p do
         /*consider each action within the delimited lifetime of X.a in p*/
         foreach action
            a := succ((→X[i].a | ↓_j X[i] | ↑_j)) .. pred((→Y.b | ↓_k X[i] | ↑_k)) do
            /*if the action can evaluate another X.a, then
               X.a cannot be a global variable*/
            case a of
               →X[j].a:    GLOBAL_VAR[X.a] := FALSE;
               ↓_k Y: if (X.a∈CAN_EVAL[Y,k]) then
                              GLOBAL_VAR[X.a] := FALSE;
               ↑_k: /*possible only if X is parent.*/
                     /*will be checked in upper production*/
                     /*if another X.a overlaps this one.*/
            end; /*case*/
```

Algorithm to Compute the Set of Attributes
Implementable as Global Variables

- Figure 2.2 -

attributes implementable as global variables. The alternative to this, pointed out in [FaY86], is to assume that $X.b$ violates the global variable property if $\downarrow_k Y$ is performed during the lifetime of $X.b$ where $Y \Rightarrow^* X$. Since attributes are only defined during **one** visit to a symbol, we can easily see that a visit to Y from X will not necessarily create a new $X.a$.

Intuitively, $CAN\_EVAL[S,v]$ is computed as follows: first, given a production

$$p: S ::= \alpha$$

with visit sequence

$$\uparrow_0, \cdots, \uparrow_1, \cdots, \uparrow_n$$

```
/*
On Exit: X.a∈CAN_EVAL[S,v] iff X.a can be evaluated during visit v to symbol
        S. */
/*initialize*/
MARKED[X,i,Y,j] := FALSE for all symbols X,Y and visits i,j;
foreach visit v to symbol S do
   MARKED[S,v,S,v] := TRUE; /*do not process recursive visits*/
   /*
     initialize CAN_EVAL by gathering all ''directly'' executed
     actions as described.
   */
   CAN_EVAL[S,v] := {};
   foreach production p with parent symbol S do
      CAN_EVAL[S,v] :=
        CAN_EVAL[S,v] ∪ {a|a is an action between $\uparrow_{v-1}, \cdots, \uparrow_v$ in p}

/*now, hoist unprocessed (unMARKED) ''indirect'' visit actions*/
while ∃ S,v,X,i such that
         action $\downarrow_i$X∈ CAN_EVAL[S,v] and NOT MARKED[S,v,X,i] do
   MARKED[S,v,X,i]:=TRUE;
   CAN_EVAL[S,v]:=CAN_EVAL[S,v] ∪ CAN_EVAL[X,i]; /*hoist actions*/
   foreach action $\downarrow_j$Y∈ CAN_EVAL[X,i] do /*hoist MARKs too*/
      if MARKED[X,i,Y,j] then MARKED[S,v,Y,j]:=TRUE;
```

- Figure 2.3: Computation of *CAN_EVAL* matrix -

all the actions between $\uparrow_{v-1} \cdots \uparrow_v$ go into *CAN_EVAL* $[S,v]$. Thereafter, given action $\downarrow_i X \in$ *CAN_EVAL* $[S,v]$, further processing is required. This action represents the fact that the TWA can visit X for the i^{th} time **during** visit v to S. Therefore, all those actions possible during visit i to X (namely those in *CAN_EVAL* $[X,i]$) must be "hoisted" up into *CAN_EVAL* $[S,v]$. If done correctly, a finite number of hoisting actions will lead to a configuration where no further changes to any *CAN_EVAL* entry is possible. At this point, the *CAN_EVAL* matrix is "complete," and can be used in the algorithm of Figure 2.2. Figure 2.3 shows how to compute the *CAN_EVAL* matrix using this method. *MARKED* $[S,v,X,i]$ tells that the visit action $\downarrow_i X \in$ *CAN_EVAL* $[S,v]$ has already caused *CAN_EVAL* $[X,i]$ to be hoisted up into *CAN_EVAL* $[S,v]$. On a practical note, the algorithm halts because of two things:

(1) No $CAN\_EVAL[X,i]$ is ever hoisted twice into a $CAN\_EVAL[S,v]$, since an action $\downarrow_i X \in CAN\_EVAL[S,v]$ is *MARKED* as soon as it is hoisted. Furthermore, the initialization of

$$MARKED[S,v,S,v] := TRUE$$

ensures that $CAN\_EVAL[S,v]$ will never be hoisted into itself.

(2) The number of visits to symbols is finite.

Moreover, note that further hoisting operations would not change any *CAN_EVAL* entry, since all visit actions in all *CAN_EVAL* entries have been *MARKED* upon exit from the *while* loop. Therefore the algorithm is correct.

Here, with the algorithm for detecting global variables fresh in mind, we point out a fruitful transformation on visit sequences. Since using a global variable is such a useful way to store an attribute, it is desirable to alter visit sequences slightly in order to get more of them. The algorithm of Figure 2.2 can *easily* be changed to record the location of those actions which cause an attribute $X.a$ to violate the global variable constraints. For those attributes which are **almost** global variables, except for one or two conflicting actions, an operation *shuffle*() might successfully remove the conflict by moving the conflicting action to some earlier or later place in the visit sequence. For example, suppose we had the AG fragment with inherited attributes $X.a, Z.b$:

$$\begin{array}{l} p: X ::= X\ Z \\ \quad r_{p,j}: X[2].a := f(\cdots X[1].a \cdots); \\ \quad r_{p,j+1}: Z.b := f(\cdots X[1].a \cdots); \end{array}$$

having visit sequence:

$$vs: \uparrow_0, \cdots, \rightarrow X[2].a, \downarrow_1 X[2], \rightarrow Z.b, \cdots$$

(assume the birth of $X[1].a$ occurs at the $\uparrow_0$ action). The occurrence of action

$\rightarrow Z.b$ after $\rightarrow X[2].a$ violates the global variable condition for $X.a$. Suppose p is the only production violating the condition for $X.a$. If $\rightarrow Z.b$ were moved to a point before $\rightarrow X[2].a$ in the visit sequence, then $X.a$ might be a global variable. In general, this move can be made whenever the following are true:

(1) It does not violate the ordering of actions required by the graph $DP(p) \cup DDP(p)$ (this graph is defined in detail in the next chapter and in [WaG83,p.198]).

(2) It does not cause $Z.b$ to violate the conditions for being a global variable, whereas without the move it could be a global variable.

(3) It does not cause $Z.b$ to become multi-visit.

As long as these conditions are not violated, we do not mind extending the lifetime of $Z.b$ for the purpose of making $X.a$ a global variable. According to the next sections, so long as $Z.b$ remains single visit, it can always be implemented as a stack, **regardless** of how much we extend its lifetime.

2.4. Implementing attributes using global stacks

Although not quite as advantageous as global variables in terms of run-time efficiency, this form of storing attributes nonetheless yields tremendous space savings over storing attributes in the structure tree [KHZ82]. Unfortunately, existing compiler writing systems fail to implement as many attributes using stacks as they might [FaY86]. As a remedy, the following sections will show how to correctly implement a significantly larger class of attributes using stacks than is currently done, and will further show how to achieve the best method for stacking that is possible for a particular attribute.

In order to easily transform a visit sequence which stores attributes in tree nodes into a visit sequence that uses stack operations, it is useful to associate an ''obituary'' and a ''birth'' record with every action in the visit sequence. For action *a* in a visit sequence, *obituary*[*a*] is the set of attributes which are guaranteed to have no further uses beyond that action. *Birth*[*a*] is the set of attributes that are guaranteed to be alive after action *a* is executed[4]. These sets make it possible for a visit sequence translation algorithm to insert stack operations into the visit sequence for a production without regard to where the production might occur in a structure tree. This makes the translation algorithm simpler.

For a visit sequence translation algorithm to work at all for the extended class of attributes we propose to deal with, the following extensions to the traditional stack operations must be provided:

```
PUSH(S,v) → S /*place v on top of S*/ -
TOP(S,i) → v
   /*return value in the ith stack position
         in S, where TOP(S,0) is the top-most value*/
POP(S,i) → S
   /*remove the ith stack element of S, where
         POP(S,0) removes the top-most stack element*/
CLOBBER(S,v,i) → S
   /*overwrite the ith stack element of S with v, where
         CLOBBER(S,v,0) overwrites the top-most stack element*/
SWAP(S,i) → S
   /*exchange the ith stack element of S with the 0th stack element*/
```

Of course, since these operations will be inserted into visit sequences statically, we must guarantee that ''constant'' offsets into the stacks (determined at compiler generation time) suffice to access any stack value needed at TWA runtime. The operations must also be inserted into correct locations of the visit sequences, such that exactly one allocation operation is guaranteed to be executed by the TWA before an attribute value is created, and exactly one deallocation operation is guaranteed to be executed for the attribute when its value is no longer needed. Insertion of a *CLOBBER* operation acts as both a deallocation routine for a dying

[4]Recall that visit sequences for simple multi-visit AGs are fixed, regardless of the context they are used in. This guarantees that *birth* and *obituary* sets can be created for any action.

attribute and an allocation routine for a created attribute. This means that the operations which can bracket the lifetime of an attribute in a correct execution of the TWA can either be *PUSH*es, *POP*s, or *CLOBBER*s. Before specifying algorithms to place these operations into visit sequences in such a way as to produce correct TWA executions, we note that such placement must differ for inherited and synthesized attributes. Therefore the following sections deal with those attributes separately.

2.4.1. Inherited single visit attributes

For this case, we insert operations to allocate and deallocate space for an attribute instance $X.b$ into the visit sequence for productions p having X as a child symbol. This is traditionally known as implementing attributes using stacks pushed and popped "from above" [FaY86]. Note that such p can have several occurrences of X, and any number of the $X[i].b$ can be alive (and on the X_b stack) when any given action a in the visit sequence is executed. As pointed out before, to "statically" implement $X.b$ as a stack, the index parameters of the stack operations applied to X_b must be constants. Therefore, at visit sequence creation time, we must keep track of which X_b stack element will correspond to which instance of $X[i].b$ after any action a has been executed at TWA runtime. To do this, each of our translation routines keeps an array called *offset*, where the value of $offset[i]$ (at generation time) is the index into the X_b stack that instance $X[i].b$ will have at TWA runtime. For example, if $offset[i]$=2 at action a="$\rightarrow Y.c$", then the rule

$$r_{p,j}: Y.c := f(\cdots X[i].b \cdots)$$

which defines $Y.c$ can be changed to

$$r_{p,j}: Y.c := f(\cdots TOP(X\_b,2) \cdots)$$

An $offset[i]=-1$ at action a means that the value of $X[i].b$ will not be on the X_b

stack when a is executed by the TWA.

The *offset* array will, of course, have to be modified after the translation of some actions, because stack operations will be emitted. The following enumerates the possible operations emitted, and the changes that have to be made to *offset* in order to correctly model the stack state:

(1) given an emitted $PUSH(X\_b, r_{pj})$ where r_{pj} computes $X[i].b$, $offset[i]$ gets the value 0, while all other $offset[j]$ with values greater than -1 are increased by one (to show their new stack depth).

(2) given an emitted $POP(X\_b, i)$ where i is constant, the *offset* entry with value i is changed to -1, and all *offset* entries with values greater than i are decreased by one to show their new stack depth.

(3) given an emitted $SWAP(X\_b, i)$ where i is constant, the *offset* entry with value i is given value 0, and the *offset* entry with value 0 is given value i. No other *offset* entries change.

(4) given an emitted $CLOBBER(X\_b, r_{pj}, k)$ where r_{pj} computes $X[i].b$ and k is constant, the *offset* entry with value k is given value -1 (it was clobbered), and the $offset[i]$ entry is given the value k (it is now using that stack entry). No other *offset* entries change.

(5) given an emitted $TOP(X\_b, i)$, no *offset* entries change.

Using the *offset* array and stacks pushed and popped from above, one can use the algorithm of Figure 2.4 to translate visit sequences for stackable inherited attributes. The following lemmas and theorem show that the algorithm statically implements inherited $X.b$ as a stack, and does so correctly, given the sufficient condition that $X.b$ is single visit.

Lemma 2.1

Given production p containing a child instance of symbol X (call it $X[i]$) with

```
Given:  visit sequence $\uparrow_0 \cdots \uparrow_1 \cdots \uparrow_m$ augmented with birth, obituary sets.
        inherited stackable attribute $X.b$
        update () which alters offset for "push","pop","swap" operations
Algorithm:
for i := 1 to |X [j]∈p | do offset[i] := -1; /*nothing stacked yet*/
for a := $\uparrow_0$ .. $\uparrow_m$ do /*foreach action in the visit sequence*/
   case a of
     "→Y.c" : foreach X[i].b referenced by $r_{p,l}$ do /*$r_{p,l}$ defines Y.c*/
                    replace "X[i].b" in $r_{p,l}$ with "TOP(X_b,k)" where k=offset[i];
                 output("→Y.c"); /*no change to action*/
                 foreach X[j].b∈obituary[a] do
                    output("POP(X_b,k)") where k=offset[j];
                    update(offset,j,"pop");
     "$\downarrow_i$ Y" : output("$\downarrow_i$ Y"); /*no action or rule changes*/
     "→X[j].b": foreach X[i].b referenced by $r_{p,l}$ do /*$r_{p,l}$ defines X[j].b*/
                      replace "X[i].b" in $r_{p,l}$ with "TOP(X_b,k)" where k=offset[i];
                    if (∃ X[i].b∈obituary[a]) then
                      output("CLOBBER(X_b,$r_{p,l}$,k)") where k=offset[i];
                      /*update(offset, "clobber")*/
                      offset[j] := offset[i]; /*new value overwrites old*/
                      offset[i] := -1; /*old value now unavailable*/
                      obituary[a] := obituary[a] - {X[i].b}
                    else output("PUSH(X_b,$r_{p,l}$)");
                       update(offset,j,"push");
                    foreach X[i].b∈obituary[a] do
                      output("POP(X_b,k)") where k=offset[i];
                      update(offset,i,"pop");
     "$\downarrow_j$X[i]": if (offset[i]<> -1) then /*X[i].b is alive*/
                    if (offset[i]<>0) then /*SWAP to stack TOP*/
                      output("SWAP(X_b,k), $\downarrow_j$ X[i]") where k=offset[i];
                      update(offset,i,"swap");
                 output("$\downarrow_j$ X[i]");
                 if (X[i].b∈obituary[a]) then
                    output("POP(X_b,0)");
                    update(offset,i,"pop");
     "$\uparrow_i$":   if (X[1].b∈birth[a]) then /*parent X.b pushed on during LEAVE*/
                 offset[1] := 0;
              if (X[1].b∈obituary[a]) then /*parent X.b popped off during LEAVE*/
                 offset[1] := -1;
              output("$\uparrow_i$");
```

Algorithm to translate visit sequences for inherited attributes using constant stack offsets

- Figure 2.4 -

single visit inherited attribute $X.b$, the algorithm of Figure 2.4 inserts exactly one stack allocation and one stack deallocation operation for $X[i].b$ into the visit sequence for p.

proof

Since we are dealing with inherited attributes here, exactly one $\rightarrow X[i].b$ must exist for each child symbol $X[i]$ in p. The code for case $a=$"$\rightarrow X[i].b$" is guaranteed to emit one allocation operation for $X[i].b$. This is the only case where an allocation operation is emitted for any $X[i].b$, so exactly one allocation is emitted for each child $X[i].b$ in any such production. As for the deallocation operation, we note that $X[i].b$ belongs to exactly one *obituary*$[a]$. It cannot belong to *obituary*$[\downarrow_j Y]$ because it cannot be used during a visit to Y. It cannot belong to *obituary*$[\uparrow_j]$ since it is single visit and $X[i]$ is a child symbol of p. All other actions a' emit exactly one *POP* or *CLOBBER* for $X[i].b \in$ *obituary*$[a']$. Hence exactly one deallocation operation is emitted for each $X[i].b$ occurrence.

□ (lemma 2.1)

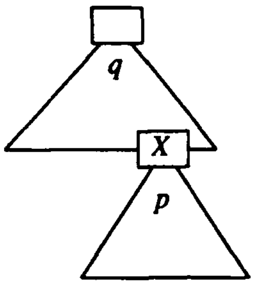
Lemma 2.2

Using the algorithm of Figure 4.2, the value of *offset*$[i]$ after each visit sequence action a for production p accurately reflects the position that attribute instance $X[i].b$ in p will have in the X_b stack after the TWA has executed that action at runtime.

proof (induction)

- Basis -

We must first show that the *offset* array is correct after execution of the first visit sequence action $a=$"$\uparrow_0$" for production p. If X is not the parent symbol for p, then Lemma 2.1 ensures that no $X[i].b$ values occurring in p can possibly be on the stack after action a. The translation algorithm leaves *offset* $[i]=-1$ for all i in this case, which is clearly correct. Now suppose X is the parent symbol of p, and that an instance of p in the structure tree is the child of some other production q, connected via symbol X as follows:



If $X[1].b \notin birth[a]$, then "*offset* $[i]=-1$ for all i" is also a correct model of the stack values at this point, since when the TWA visits p from q no instances of $X[i].b$ from p are on the stack. However, if $X[1].b \in birth[a]$, then the TWA must place $X[1].b$ in the X_b stack before making the visit from q to p. The code emitted for case "$\downarrow_1 X$" in q's visit sequence ensures that this will happen. The translation algorithm sets *offset* $[1]=0$ when processing p, which accurately reflects the runtime state of the X_b stack after execution of action a.

- Inductive Step -

Assume processing the k^{th} action a^k has left the *offset* array in a correct configuration. We consider each possible $k+1^{st}$ action a^{k+1} in turn:

"$\rightarrow Y.c$": leaves *offset* in a correct configuration since no $X[i].b$ values can be created, and all child $X[i].b$ last used here are popped off, which is reflected by manipulation of *offset* by *update*.

"$\downarrow_i Y$": since $X.b$ is single visit, any instances of $X.b$ created while TWA visits Y will be dead before TWA executes the $\uparrow_i$ action returning it from Y.

The placement of single visit attributes in *obituary* sets and Lemma 2.1 ensures that dead $X.b$ are popped off before *LEAVE* actions, so any $X.b$ values pushed onto X_a while TWA processes Y's subtree are popped off before TWA returns. No modifications to *offset* are needed for this case.

"$\uparrow_i$": since $X.b$ is single visit, no child instances of $X[i].b$ are on the stack before or after this *LEAVE* operation. We therefore know that their *offset* values are -1. For the parent instance, using the same argument as for the basis, if $X[1].b$ is created during this leave, it will be on the top of the X_b stack if it is still alive upon return. This is correctly reflected by the assignment to *offset* given for this case. Likewise, if $X[1].b$ happens to die during this *LEAVE*, we know its *POP* or *CLOBBER* will be inserted in the upper visit sequence, so all we need to do to *offset* after a is to reflect that **no** $X[i].b$ values from this production are on the X_b stack, which the assignment $offset[1]{:=}-1$ does.

"$\rightarrow X[i].b$": the changes to *offset* here are correct since the one created $X[i].b$ is allocated space, and since all dead $X[j].b$ (including $X[i].b$ if it is not used anywhere) are popped off.

"$\downarrow_j X[i]$": if $X[i].b$ is alive for this visit, then the swap places it on top of the stack, which is reflected in the assignment to *offset*. If the attribute is last used in the visit, then we know it is still on the top of stack upon return (see argument for "$\downarrow_i Y$" case), so popping the top of stack is guaranteed to work. This *POP* is correctly reflected in the assignment to *offset*. Since $X.b$ is inherited, if $X[i].b$ is not alive before the visit, then it is also not alive after the visit, so no need to modify *offset* for a *PUSH*. Finally, using the same argument as for the case a="$\downarrow_j Y$", any new $X.b$ instances pushed on during the visit are popped off before control returns. Hence we see that the stack configuration before the visit is the same as after the visit. Since no changes are made to *offset* in this case, it correctly reflects the state of the X_b stack after the visit.

□ (lemma 2.2)

Lemma 2.3

The algorithm of Figure 4.2 replaces any use of $X[i].b$ in any semantic rule of a production by $TOP(X\_b,k)$ where k is the correct offset of the $X[i].b$ value in the X_b stack when the rule is executed.

proof

We can easily see that the only uses of $X[i].b$ in a visit sequence being processed occur at the actions $\rightarrow Y.c$, $\rightarrow X[j].b$. For these actions, the *foreach* loop guarantees that the rules referring to $X[i].b$ are all modified with the TOP operations.

□ (lemma 2.3)

Theorem 2.1

Any single visit inherited attribute $X.b$ can be statically implemented using a stack pushed and popped from above.

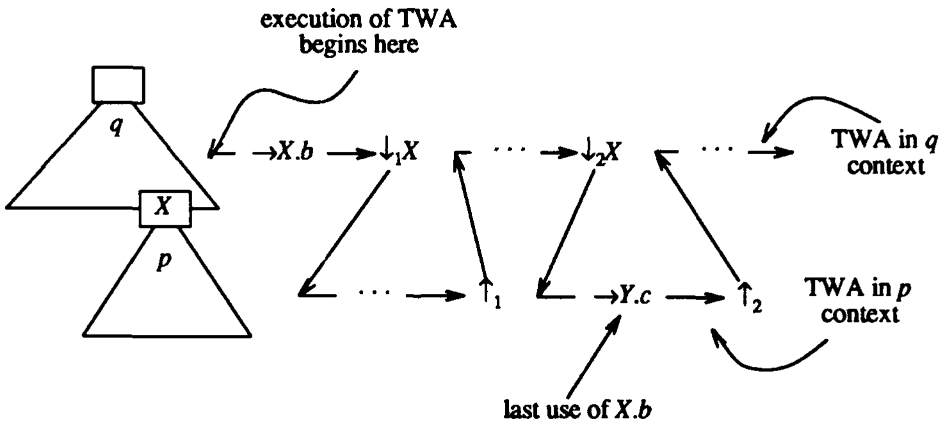
proof (using algorithm Figure 2.4)

Lemma 2.1 ensures that the TWA will encounter exactly one allocation and exactly one deallocation operation for each instance of a single visit inherited attribute. Lemma 2.2 shows that the constant offsets that the TWA will use when it encounters these allocation and deallocation routines are correct. Lemma 2.2 and Lemma 2.3 together show that the references to inherited single visit attributes in the semantic rules are correctly translated into references to stack elements with the proper constant offsets. This proves the theorem.

□ (theorem 2.1)

2.4.2. Inherited single visit attributes last used during visits

Consider an inherited attribute $X.b$ which **always** dies during some visit to its nonterminal symbol X. When implementing $X.b$ as a stack, an operation which allocates space for an instance of $X.b$ can be placed in all productions where X is a child symbol, and a deallocation operation can be inserted in all productions where X is the parent symbol. This will work correctly since every appearance of a nonterminal X in a structure tree is guaranteed to connect two productions p and q, where X is a child symbol of q and the parent symbol of p, as follows:



Inserting allocation and deallocation operations as described above, execution of the TWA in a q,p context is guaranteed to allocate and deallocate space for inherited $X.b$. This matched allocation and deallocation is graphically depicted using the TWA "execution path" above, if the $X.b$ value is pushed on where it is created in q and popped off where it dies after defining $.b$ in p.

In [FaY86], they argue convincingly for using this method to stack inherited attributes where possible. Briefly, they claim that inserting allocation and deallocation routines as "close as possible" to where attributes are defined and last used helps keep stacks significantly smaller than when they are inserted otherwise. This makes sense because allocating space for an attribute "too soon" or "too late" means that the space might go unused indefinitely while the TWA executes actions unrelated to the attribute.

In order to take advantage of the economy of "close as possible" stacking, this section will show how to correctly implement **any** inherited single visit attribute (which always dies during a visit to its symbol) as a stack pushed from above and popped from below.

It should be pointed out in passing that this is a significant class of attributes; specifically, any inherited attribute obeying Bochmann Normal Form (BnNF) [Boc76] always dies during a visit to its symbol. Furthermore, although BnNF precludes the "use" of an inherited attribute in the production where it is defined, this thesis allows such use, so long as the attribute eventually dies during a visit to its symbol.

The algorithm of Figure 2.5 shows how to insert push from above and pop from below stack operations into visit sequences to implement inherited attributes as stacks. In order to work, the algorithm demands that all child instances of attribute $X.b$ considered for this stacking method must appear in the *obituary* set of some action "$\downarrow_i X$". In addition, the properties of $X.b$ must be such that the *offset* array is correctly maintained. The following theorem gives a sufficient condition for $X.b$ to guarantee this correct maintenance.

Theorem 2.2

Given any production p containing X as a child symbol, if the last use of single visit inherited attribute $X.b$ is **always** during some $\downarrow_i X$, then $X.b$ can be implemented as a stack pushed from above and popped from below.

```
Given:  visit sequence $\uparrow_0 \cdots \uparrow_1 \cdots \uparrow_m$ augmented with birth, obituary sets.
        inherited stackable attribute $X.b$ whose last use is always during
            some $\downarrow_i X$.
        update() which alters offset for "push","pop","swap" operations.
Algorithm:
for i := 1 to |X [j]∈p | do offset[i] := -1; /*nothing stacked yet*/
for a := $\uparrow_0$ .. $\uparrow_m$ do /*foreach action in the visit sequence*/
    case a of
      "→Y.c" : foreach X[i].b referenced by $r_{p,l}$ do /*$r_{p,l}$ defines Y.c*/
                  replace "X[i].b" in $r_{p,l}$ with "TOP(X_b,k)" where k=offset[i];
               if (X[1].b∈obituary[a]) then /*only parent X.b dies here*/
                  output("POP(X_b,k)") where k=offset[1];
                  update(offset,1,"pop");
      "$\downarrow_i$ Y" : output("$\downarrow_i$ Y"); /*no action or rule changes*/
      "→X[j].b" : foreach X[i].b referenced by $r_{p,l}$ do /*$r_{p,l}$ defines X[j].b*/
                    replace "X[i].b" in $r_{p,l}$ with "TOP(X_b,k)" where k=offset[i];
                 if (X[1].b∈obituary[a]) then
                    output("CLOBBER(X_b,$r_{p,l}$,k)") where k=offset[1];
                    /*update(offset, "clobber")*/
                    offset[j] := offset[1]; /*new value overwrites old*/
                    offset[1] := -1; /*parent value now unavailable*/
                 else output("PUSH(X_b,$r_{p,l}$)");
                    update(offset,j,"push");
      "$\downarrow_j$X[i]" : if (offset[i]<> -1) then /*X[i].b is alive*/
                     if (offset[i]<>0) then /*must swap to make X[i].b stack top*/
                        output("SWAP(X_b,k), $\downarrow_j$X[i]") where k=offset[i];
                        update(offset,i,"swap");
                  output("$\downarrow_j$X[i]");
                  if (X[i].b∈obituary[a]) then /*dies during visit*/
                     /*POP is emitted in VS below*/
                     update(offset,i,"pop");
      "$\uparrow_i$": if (X[1].b∈birth[a]) then /*parent X.b pushed on during LEAVE*/
                offset[1] := 0; /*parent instance value is on stack top*/
             output("$\uparrow_i$");
```

Algorithm to translate visit sequences for inherited attributes
(which always die during visits to their symbol)
using constant stack offsets
- Figure 2.5 -

proof (using algorithm Figure 2.5)

The proof of this theorem is very similar to the proof of Theorem 2.1, and so will not be given in detail here. Instead, it suffices to point out the differences between this method of stacking and the method of Theorem 2.1, and show that those differences are correctly handled by the algorithm of Figure 2.5.

(1) In the pop from above method, the translation algorithm must handle any number of child $X.b$ instances which die. In this method, only one instance (the parent $X.b$) can die (all child $X.b$ instances die during some "$\downarrow_i X[j]$"). Therefore, in those places (using the pop from above method) where *POP* operations must be inserted for **every** dying child $X.b$, only **one** *POP* must be inserted for possibly dying parent $X.b$ (using this method). This is accurately reflected by the emission of a single deallocation for the parent $X[1].b$ in the only places $X[1].b$ can be used: "$\rightarrow Y.c$" and "$\rightarrow X[i].b$".

(2) In the pop from below method, when the TWA executes action "$\uparrow_i$", it cannot *POP* the parent $X[1].b$ off the stack (since we are now using pop from below). No code to keep track of a *POP* here is given in the new algorithm. However, the parent instance $X[1].b$ might still get pushed on here, so for this case the code remains the same.

(3) In the pop from below method, when the TWA executes action "$\downarrow_j X[i]$", instance $X[i].b$ might get popped off in the lower visit sequence. Hence we do not output a *POP* operation here, but must keep track of the possibility of the *POP* in the lower visit sequence.

□ (theorem 2.2)

2.4.3. Synthesized single visit attributes

Unlike inherited attributes, synthesized attributes are always defined when the TWA is visiting a production instance for which the attribute's symbol is the **parent** node. Also unlike inherited stackable attributes, which sometimes must be pushed and popped in the same (upper) visit sequence, synthesized stackable attributes can **always** be pushed in the lower and popped in the upper visit sequence. Intuitively this makes sense; if the TWA could deallocate space for the attribute in its lower production context, it is **guaranteed** to visit the attribute's upper production context again, where the deallocation could instead be performed.

With this in mind, we provide an algorithm in Figure 2.6 for stacking synthesized single visit $X.b$. As before, it must be shown that the algorithm correctly adheres to the stack discipline, and that the constants maintained by the *offset* are correct. Theorem 2.3 provides a sufficient condition for synthesized $X.b$ to guarantee the correctness of this algorithm.

Theorem 2.3

Any single visit synthesized attribute can be statically implemented as a stack pushed from below and popped from above.

proof (using algorithm Figure 2.6)

Again, this algorithm is similar enough to that in Figure 2.5 (inherited pushed from above and popped from below) that we need only point out the differences between the two methods and show how the new algorithm accounts for those differences.

(1) For the inherited scheme, any number of child $X.b$ instances can require pushing in the visit sequence. In the synthesized scheme, only the parent $X.b$ instance can require pushing. This is reflected by the single case "$\rightarrow X[1].b$" in the synthesized algorithm, and the code therein.

```
Given:  visit sequence $\uparrow_0 \cdots \uparrow_1 \cdots \uparrow_m$ augmented with *birth*, *obituary* sets.
        synthesized stackable attribute *X.b*
        *update* () which alters *offset* for "push","pop","swap" operations.
Algorithm:
for i := 1 to $|X[j] \in p|$ do offset[i] := -1; /*nothing stacked yet*/
for a := $\uparrow_0 .. \uparrow_m$ do /*foreach action in the visit sequence*/
  case a of
    "$\rightarrow$Y.c" : foreach X[i].b referenced by $r_{p,l}$ do /*$r_{p,l}$ defines Y.c*/
                 replace "X[i].b" in $r_{p,l}$ with "TOP(X_b,k)" where k=offset[i];
               output("$\rightarrow$Y.c"); /*no change to action*/
               foreach X[i].b$\in$obituary[a] do
                 output("POP(X_b,k)") where k=offset[i];
                 update(offset,i,"pop");
    "$\downarrow_i$ Y" : output("$\downarrow_i$ Y"); /*no action or rule changes*/
    "$\rightarrow$X[1].b" : /*only parent defined here; $r_{p,l}$ defines X[1].b*/
             foreach X[i].b referenced by $r_{p,l}$ do
               replace "X[i].b" in $r_{p,l}$ with "TOP(X_b,k)" where k=offset[i];
             if ($\exists$ X[i].b$\in$obituary[a]) then
               output("CLOBBER(X_b,$r_{p,l}$,k)") where k=offset[i];
               /*update(offset, "clobber")*/
               offset[1] := offset[i]; /*parent value overwrites child*/
               offset[i] := -1; /*overwritten value now unavailable*/
               obituary[a] := obituary[a] - {X[i].b};
             else output("PUSH(X_b,$r_{p,l}$)");
                update(offset,1,"push");
             foreach X[i].b$\in$obituary[a] do
               output("POP(X_b,k)") where k=offset[i];
               update(offset,i,"pop");
    "$\downarrow_j$X[i]" : if (offset[i]<> -1) then /*X[i].b is alive*/
                   if (offset[i]<>0) then /*must swap to make X[i].b stack top*/
                     output("SWAP(X_b,k), $\downarrow_j$X[i]") where k=offset[i];
                     update(offset,i,"swap");
                output("$\downarrow_j$X[i]");
                if (X[i].b$\in$birth[a]) then /*defined and PUSHed during visit*/
                   update(offset,i,"push");
                if (X[i].b$\in$obituary[a]) then /*died during visit; POP here*/
                   output("POP(X_a,k)") where k=offset[i];
                   update(offset,i,"pop");
    "$\uparrow_i$":  if (X[1].b$\in$obituary[a]) then /*parent X.b popped off during LEAVE*/
               offset[1] := -1;
             output("$\uparrow_i$");
```

Algorithm to translate visit sequences for synthesized attributes using constant stack offsets

- Figure 2.6 -

(2) For the inherited scheme, only the parent instance of $X.b$ can be popped off after a use in the visit sequence. In the synthesized scheme, any number of child instances of $X.b$ can be popped after a use in the visit sequence. This is reflected in the synthesized algorithm by popping **all** child instances dying after a use (cases "$\rightarrow Y.c$", "$\rightarrow X[i].b$", and "$\downarrow_i X[j]$").

(3) Child inherited instances can be popped off during action "$\downarrow_j X[i]$", while child synthesized instances can be pushed on then. The code for this case in the synthesized translator reflects this difference.

(4) Parent inherited instances of $X.b$ can be pushed on during "$\uparrow_i$" actions, while parent synthesized instances can be popped off then. The code for synthesized translation in this case reflects the difference. □ (Theorem 2.3)

2.4.4. Performance note on stacking

In the spirit of performing ‘‘close as possible’’ stacking for synthesized attributes also, we could give a translation algorithm which handles the case where a synthesized attribute always dies during a visit to its symbol (just as was done for inherited). However, while the argument for doing so with inherited attributes is strong (all BnNF inherited attributes can be translated this way), it is weak for synthesized attributes (BnNF synthesized attributes are best stacked via Figure 2.6). Therefore we choose not to develop such an algorithm.

A different optimization issue involving these algorithms is the number of times a *CLOBBER* is used instead of a *POP* followed by a *PUSH*. The use of the *CLOBBER* is, of course, less time consuming, but there is an another good reason for its use: inserting a *CLOBBER* where the semantic rule involved has the form:

$r_{p,i}: X[i].b := X[j].b;$

means that the *CLOBBER* can be left out all together! This is historically known as "copy rule elimination" [FaY86]. To get the maximum amount of *CLOBBER*s (and thereby more copy rule eliminations), if the last use of an $X[i].b$ is at some $\rightarrow Y.c$ action, it would be better not to place $X[i].b$ into this action's obituary if some action $\rightarrow X[j].b$ is only a "little" later in the visit sequence. $X[i].b$ would be better placed into *obituary*$[\rightarrow X[j].b]$, since a *CLOBBER* would then result. By "little" we mean that the placement should not make $X[i].b$ a mutli-visit instance. Also, it might be even better, if there is a choice of $X[j].b$ obituaries to place $X[i].b$ in, to use the one involved in a copy rule with $X[i].b$. Of course, this artificial extension of lifetimes runs somewhat against the idea of "close as possible" stacking, but better runtime results might be shown to outweigh the additional space costs.

2.5. Practical runtime optimization

Given that some attributes must remain structure tree resident, some common sense strategies should be observed when implementing them as such. The current strategy of the GAG compiler generator system [KHZ82] is to allocate the structure tree node and its corresponding attribute storage during parse time, when a production is reduced. Deallocation occurs only after all attribution has been performed. There are many ways to optimize this approach to attribute storage.

As a first optimization, the storage for all the child nodes of a production can be reclaimed when its visit sequence calls for the final *LEAVE*. Suppose the amount of storage locations needed to store the structure tree nodes and all attributes is N. A GAG evaluator will require extra storage as it runs, since many attributes are dynamic *LIST*s. Suppose the amount of storage locations needed to store all dynamic attributes is M. As they are now implemented, GAG evaluators require $N+M$ storage locations to process a typical source input. However, deallocating storage for nodes "on the fly" will keep the total storage requirements between N and $N+M$, since some structure tree nodes will be deallocated before some dynamic attributes are ever evaluated.

As a second optimization, attribute storage could be allocated and deallocated only when needed. The visit sequences give an exact account of the lifetimes of attributes, and so a simple pair of *CREATE* and *DESTROY* operations inserted into them for each attribute would exactly match the actual storage required to process a source input. This would be optimal, and would even obviate the need for globally storing attributes, but the runtime cost of allocating and deallocating storage for each attribute instance would be much higher than using global stacks and variables. An alternative to this is just to allocate and deallocated those attributes which are left in the tree. If their number is small, then the additional cost for the allocation and deallocation might be transparent. But there remains an additional problem of how to associate a particular attribute instance with a production instance. There are several ways to accomplish this:

(1) Every structure tree node could contain storage for the root of a balanced binary tree. This tree could be keyed by attribute names, and each node could contain the storage for the attribute. This would require three extra storage locations for each attribute: one to store key, and two to store the tree links. Moreover, the access path required to obtain the proper attribute's storage depends on the depth of the binary tree, and could add substantial runtime to the attribute evaluator. This would therefore be practical only when the number of non-global attributes per symbol is very small.

(2) A hashing strategy could be adopted which maps a structure tree node's address and an attribute name to a storage location. However, making the

hash table very large to avoid collisions would be worse than storing attributes in the structure tree, and so is unacceptable. On the other hand, making the hash table very small would destroy the speed of the access functions, and so is also undesirable. Therefore hashing is not an acceptable method for mapping attribute instances to storage locations.

(3) Apparently, the most practical method would be to allocate an ''attribute area pointer'' in each structure tree node at parse time. Then, when a node's first attribute is evaluated during evaluation time, **all** attribute storage for that node is allocated in one lump and is attached to the node via the pointer (note that an action to perform this allocation can be inserted into visit sequences at compiler generation time). This yields an access path for each attribute which is only one level of indirection worse than allocating nodes with attribute storage built in. Moreover, since the storage is allocated approximately ''on demand,'' the instantaneous requirements could be much better. This avoids the increased storage and access time of binary trees, while giving a good ''fit'' to the actual storage required to process a typical source input.

The third strategy could be made even better by overlaying the storage in an attribute area for two attributes which have non-overlapping lifetimes. However, since the usual number of visits to a node is two or three [KHZ82], this optimization is probably not worth implementing.

On a practical note, if only one attribute is tree resident for a symbol, that attribute can be stored in the pointer location reserved for the attribute area address.

2.6. Conclusions

The method for defining a single visit attribute in this thesis is both more uniform and more general than in past works [FaY86, Saa78]. This is significant since the thesis further shows all single visit attributes are implementable globally.

The idea to use global information to compute the set of attributes implementable as global variables was proposed earlier by [FaY86]. However, they proposed computing a *CAN_REFERENCE* matrix which, unlike the *CAN_EVAL* matrix given here, computes all possible "uses" of attributes, in addition to just "definitions." However, the extra computation required by their method does not provide more power. In addition, they did not provide algorithms for computing either the set of global variables or the *CAN_REFERENCE* matrix.

The attributes implementable using global stacks via the algorithms given here represent a much larger class than in the previous literature. Moreover, the "close as possible" strategies of Figures 2.5 and 2.6 do not trade off the run-time stack size for the number of stackable attributes. The class implemented using close as possible stacking is the set of **all** single visit attributes in the synthesized case, and a **significant** number of stackable attributes (a superset of BnNF single visit attributes) in the inherited case.

The practical observations made here concerning the implementation of tree-based attributes, specifically concerning when storage for an attribute should be allocated and deallocated, has been skirted in the open literature. This might be because the answer appears obvious, but in fact existing compiler generator systems allocate attribute storage at parse time. The most reasonable solution to this problem appears to be to allocate all attribute storage for a structure tree node when the first attribute for that node is evaluated, and to deallocate all the storage for a node when its parent symbol's visit sequence calls for a final *LEAVE*. Once again, operations to allocate tree-based attribute storage and operations to deallocate structure tree node storage can be statically inserted into the visit sequences.

Perhaps not the least significant of the contributions of this chapter are the correct algorithms which actually translate ''inefficient'' visit sequences into those which use global stacks and global variables. As reported by the architect of the GAG system, the current algorithms for global implemention in GAG are quite ad hoc [Kas86].

CHAPTER III

CREATION OF VISIT SEQUENCES FOR BETTER ATTRIBUTE STORAGE

3.1. Motivation

In Chapter II it was shown how any single visit attribute can be implemented using a global stack or global variable. Since the single visit property for an attribute is determined solely according to the visit sequences the attribute appears in, an obvious optimization we can now make is to focus on the way visit sequences are created and, if possible, improve the number of attributes having the single visit property.

3.2. Partitions and visit sets

Consider the compiler generator's job of creating visit sequences for the productions of an attribute grammar. The same symbol X can be the parent and child symbol of many different productions:

$$\begin{array}{l} p_1 : S_i ::= \alpha_1 X \beta_1 \\ p_2 : S_j ::= \alpha_2 X \beta_2 \\ \quad\vdots \\ p_n : S_k ::= \alpha_n X \beta_n \\ \hline q_1 : X ::= \gamma_1 \\ q_2 : X ::= \gamma_2 \\ \quad\vdots \\ q_m : X ::= \gamma_m \end{array}$$

Accordingly, many different combinations of productions can be connected by the symbol X in a structure tree. Suppose the compiler generator places action $\downarrow_k X$ in the visit sequence for p_i. When inserting the remaining actions into p_i, it must know what new attributes of X are available after the TWA has executed that visit action. Since it is building visit sequences statically, the compiler generator must assume that the same set of X's synthesized attributes will be evaluated by the TWA after $\downarrow_k X$ **regardless** of which q_j is used to derive X at compiler runtime. Suppose further that the compiler generator places action $\uparrow_k$ into the visit sequence for q_j. The compiler generator must again assume that the same set of X's inherited attributes have been evaluated after the TWA executes this action, **regardless** of which p_i is the parent of q_j in the structure tree. This requirement allows us to conveniently group the attributes of a symbol according to the leave (visit) for which they are evaluated.

Definitions:

3.1 (partition)

The "partition" $\Pi^I_{X,i}$ denotes those attributes $a \in AI(X)$ which are evaluated by the TWA during its execution of the action $\uparrow_{i-1}$ for a production with parent symbol X. Similarly, the partition $\Pi^S_{X,i}$ denotes those attributes $a \in AS(X)$ which are evaluated by the TWA during its execution of the action $\downarrow_i X$ for a production with child symbol X. We note that $\Pi^I_{X,i} \cap \Pi^I_{X,j} = \emptyset$ for all $i \neq j$ (likewise for $\Pi^S_{X,i} \cap \Pi^S_{X,j}$).

3.2 (visit sets)

We conveniently group $\Pi^I_{X,i} \cup \Pi^S_{X,i}$ into the set $\Pi_{X,i}$, which we call the i^{th} "visit set" for X.

3.3 (IDS graphs)

For simple multi-visit attribute grammars, directed acyclic graphs of *I*nduced *D*ependecies within each *S*ymbol (*IDS* graphs) [WaG83, p. 195] exist which impose an order on the evaluation of X's attributes for any production where X appears.

3.4 (defining, induced, unqualified edges)

the "defining" edge $a \rightarrow_d b \in IDS(X)$ exists whenever $X.a$ is referenced in any semantic rule defining $X.b$. The "induced" edge $a \rightarrow b$ is any non-defining edge in $IDS(X)$. In situations where it is unimportant to distinguish between defining and induced edges, we will denote the edge by $a \rightarrow_u b$, and call it an "unqualified" edge.

3.5 (path)

a "path" $a \rightarrow^+ b$ represents a connected sequence of unqualified edges from a to b in $IDS(X)$. We note that, in general, a path is not unique. When we wish to indicate a particular path, it will be denoted by $a \rightarrow_i^+ b$.

Properties:

3.1 (transitive closure $IDS(X)$)

$IDS(X)$ has the property that if any $a \rightarrow^+ b$ exists, then $a \rightarrow_u b \in IDS(X)$ [WaG83, p. 195].

3.2 (visit sets)

Let $a \rightarrow_u b$ be an unqualified edge in $IDS(X)$. Each visit set $\Pi_{X,i}$ must obey the following properties:

(1) Given $a \in AS(X)$, $b \in AI(X)$, $a \in \Pi_{X,i}$, and $a \rightarrow_u b$, then $b \in \Pi_{X,k}$, $k > i$.

(2) Given $a \rightarrow_u b$, $a \in \Pi_{X,i}$ then $b \in \Pi_{X,k}$, $k \geq i$.

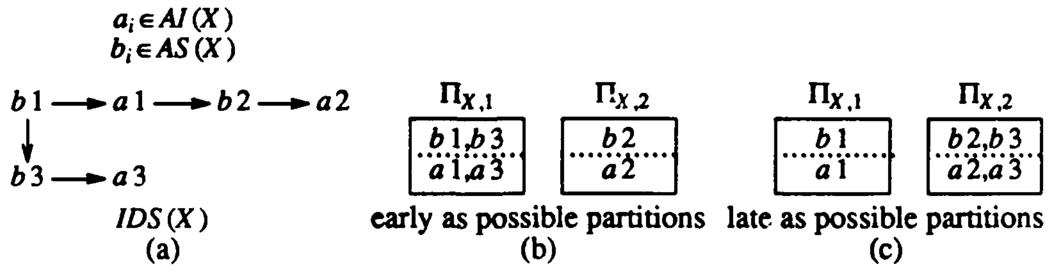
Intuitively, condition (1) of the visit set property says that if a synthesized attribute $X.a$ is evaluated during some visit to X, then all inherited attributes which "depend" on $X.a$ cannot also be computed during that visit. Condition (2) says that no attribute can be placed in an earlier partition than any attribute it depends on. Otherwise, so long as they satisfy condition (2), synthesized attributes which depend on synthesized, inherited which depend on inherited, and synthesized which depend on inherited **can** be placed in the same visit set.

We note that visit sets $\Pi_{X,i}$ for a particular $IDS(X)$ are not unique. Figure 3.1 provides an example where two visit sets satisfy the same IDS graph.

3.3. Visit sequence creation

Briefly, a visit sequence is created for a production p by performing a "topological sort" on a "visit sequence graph" that is created via analysis of the semantic rules for p and the partitions of the symbols of p. The visit sequence graph is historically known as $DP(p)$ [WaG83, p.198], and typically is a only a "partial" ordering on the actions that must occur in the visit sequence for p. Many tradeoffs exist concerning exactly where an action should be placed when transforming $DP(p)$ into a "totally" ordered visit sequence for p.

A rough outline of the steps used in producing $DP(p)$ starts with the creation of the graphs $IDS(X)$ and $NDDP(p)$ for each symbol and production (briefly, $NDDP(p)$ is the transitive closure of the dependencies induced by the semantic rules of production p). Then partitions $\Pi^I_{X,i}, \Pi^S_{X,i}$ are created for each X. Next, $DP(p)$ is created from $NDDP(p)$, $IDS(X)$, and each $\Pi_{X,i}$ such that X occurs in p. The partitions of X are included in this computation because they induce new dependencies that the TWA must abide by. For example, if $b \in \Pi^I_{X,i}$ and $a \in \Pi^S_{X,j}$



- Figure 3.1: partitions for IDS graph are not unique -

where $i \le j$, then $X.b$ must be computed before $X.a$ in any production context, so $X.b \rightarrow X.a$ is added to $DP(p)$. Now, each $DP(p)$ represents a partial ordering of the actions the TWA will execute as it walks through an instance of production p in some structure tree. All that remains is to perform a topological sort on $DP(p)$, yielding a total ordering of TWA actions. This total ordering is the visit sequence for p.[5]

The following features of this process are observable:

(1) The graphs $IDS(X)$ and $NDDP(p)$ are unique for a given AG.

(2) The graph $DP(p)$ is unique given visit sets $\Pi_{X,i}$ (and IDS, $NDDP$ graphs).

From these observations we deduce that there are only two steps in the creation of a visit sequence that we can investigate for storage optimizations: the creation of partitions, and the topological sort on $DP(p)$. The rest of this chapter will explore the impact these two steps have on making attributes single visit, and will improve the strategies currently used for them.

3.4. Impact of partitioning strategies

To see where a particular method for computing partitions can cause poor attribute storage, consider the replacement of the edge $b_1 \rightarrow b_3$ in Figure 3.1a with $b_1 \rightarrow_d b_3$. Using the visit sets of Figure 3.1c, this means the lifetime of attribute $X.b_1$ for any visit sequence must begin before the action $\downarrow_1 X$ and end no sooner than $\downarrow_2 X$ (where $X.b_3$ is evaluated). A possibility exists here for $X.b_1$ to be multi-visit. If X is a child symbol of some production p having visit sequence

$$\cdots \downarrow_1 X \cdots \uparrow_i \cdots \downarrow_2 X \cdots$$

then $X.b_1$ violates the single visit property.

[5]For a complete review of this entire process, the reader is referred to [WaG83, pp. 193-196].

Definition 3.6: (spanning attribute)

An attribute $X.a$ is called "spanning" whenever

$$a \rightarrow_d b \in IDS(X), a \in \Pi_{X,i}, b \notin \Pi_{X,i}.$$

We see that if the partitioning strategy of Figure 3.1b were used to create the visit sets for X, then $X.b_1$ would not be spanning ($X.b_3$ would be evaluated before the first visit to X) and so $X.b_1$ would be single visit for p. The "early as possible" partitioning strategy used to create these visit sets is implicit in evaluators for the "alternating" attribute grammars (AAG), suggested in [JaP75] and implemented in the Linguist-86 system [Far82]. The "late as possible" partitions of Figure 3.1c are implicit for the "ordered" attribute grammars (OAG) of [Kas80], and are implemented in the GAG system [KHZ82]. Although the AAG strategy provides optimal visit sets for this particular example, [FaY86] shows that it, too, fails to work in other situations. We therefore search for ways to reduce the number of spanning attributes in the hopes of reducing the number of multi-visit attributes.

3.5. "Near" partitions

The problem with the AAG and OAG partitioning strategies is that no attempt is made to evaluate one attribute in the same visit as another that defines it. We can correct this indifference by creating the following visit sets:

Definition 3.7: "Near" Partitioning:

given the graph $IDS(X)$, "near" visit sets for X ($\Pi^N_{X,i}$) are those which minimize:

$$\{a \mid a \in \Pi^N_{X,k} \text{ and } \exists b \notin \Pi^N_{X,k} \text{ where } a \rightarrow_d b \in IDS(X)\}$$

Although simple to express, it is not presently known if a polynomial time algorithm exists which solves this problem. However, the remainder of this section develops a heuristic that performs optimally for many instances of the problem.

Definitions:

3.8 (minimal exclusive pair)

an "exclusive pair" of attributes are any synthesized $X.a$ and inherited $X.b$ such that the edge $a \rightarrow_u b$ exists in $IDS(X)$. A "minimal exclusive pair" of attributes a,b are an exclusive pair such that no path $a \rightarrow^+ b$ exists other than $a \rightarrow_u b$ (see Figure 3.2).

3.9 (forced span)

an attribute $c \in A(X)$ is a "forced span" if there exists $a,b,d \in A(X)$ where a,b are an exclusive pair, $c \rightarrow_d d \in IDS(X)$, and $c \rightarrow^+ a \rightarrow^+ b \rightarrow^+ d \in IDS(X)$ (see Figure 3.2).

3.10 (contingent span)

an attribute $c \in A(X)$ is a "contingent spanning attribute" (or contingent attribute) if it is one of a set of attributes $c_i \in A(X)$, none of which is a forced span, but of which at least one must be spanning in legally constructed visit sets for X (see Figure 3.2).

3.11 (minimum sacrifice set)

a contingent attribute is "sacrificed" when it becomes a spanning attribute in a legal set of partitions. The "minimum sacrifice set" is the smallest set of contingent attributes that must be sacrificed in the construction of legal visit sets.

Figure 3.2 gives an example of an $IDS(X)$ containing an exclusive pair, forced span, and contingent span (in order to construct legal partitions from this graph, either of b_3,b_4 must be sacrificed). Obviously, forced spans and exclusive pairs cannot be remedied by any partitioning strategy. Therefore the problem of finding the "near" visit sets reduces to finding the minimum sacrifice set for an arbitrary

$$\begin{tikzcd}
b_2 \arrow[dr] \arrow[drr, "d"] & & \\
IDS(X) & a \arrow[r] & b_1 \\
b_3 \arrow[ur] \arrow[r, "d"] & b_4 \arrow[ur, dashed, "d"] &
\end{tikzcd}$$

$b_i \in AI(X)$

$a \in AS(X)$

a, b_1 are an exclusive pair
b_2 is a forced span
b_3, b_4 are contingent spans

- Figure 3.2 -

$IDS(X)$, and then finding partitions which embody only those sacrifices.[6]

3.5.1. Network flow heuristic

Figure 3.3a contains a more complicated $IDS(X)$ graph which illustrates the difficulty of finding the minimum sacrifice set. Intuitively, we solve the problem for this instance of $IDS(X)$ by drawing a "cut" line across the graph, as in Figure 3.3b, which divides the attributes so that the exclusive pair a, b_1 ends up on either side. This cut line induces a "canonical partitioning" of the attributes into two visit sets:

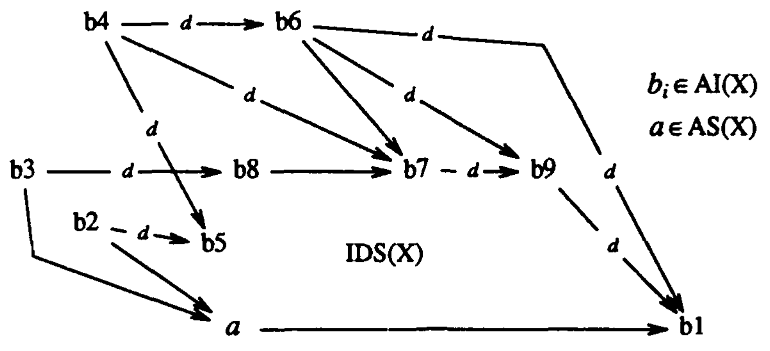
$c \in \Pi_{X,1}$ if $c \in A(X)$ is to the **left** of the cut line,

$c \in \Pi_{X,2}$ if $c \in A(X)$ is to the **right** of the cut line.

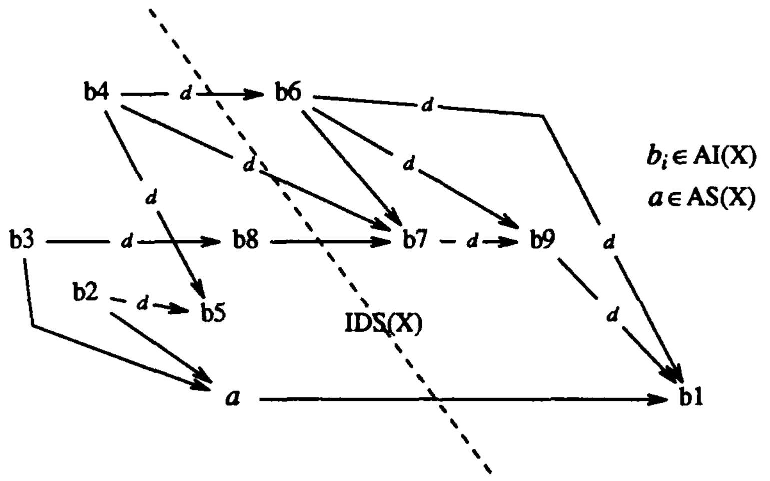
For the $IDS(X)$ graph of Figure 3.3, which has but a single minimal exclusive pair, these two visit sets completely satisfy Property 3.2 (visit sets).

Of course, since we seek visit sets which minimize the number of spanning attributes, we must use some intelligence when inserting a cut line into an

[6]We show later how to build visit sets, given a sacrifice set, such that only those attributes in the sacrifice set are spanning.



(a) - complex sacrificing problem



(b) - cut line inducing optimal canonical visit sets

- Figure 3.3 -

$IDS(X)$ graph. Fortunately, algorithms in the domain of network flow theory are guaranteed to perform **optimal** cut line insertion (in polynomial time) for graphs that look very much like the one in Figure 3.3a. We can borrow these well understood and well optimized algorithms to produce better storage optimizations for AGs.

Definition 3.12: Maximum Network Flow (from [Tar83])

Let $G = [V,E]$ be a directed graph with two distinguished vertices, a "source" s and a "sink" t, and a positive capacity $cap(vw)$ on each edge vw. If no edge vw exists, then $cap(vw)=0$. A *flow* in G is a real-valued function f on vertex pairs having the following properties:

(1) "Skew symmetry": $f(vw)=-f(wv)$. If $f(vw)>0$, there is flow from v to w.

(2) "Capacity constraint": $f(vw)\le cap(vw)$. If vw is an edge such that $f(vw)=cap(vw)$, the flow "saturates" vw.

(3) "Flow conservation": for every vertex v other than s and t, $\sum_w f(vw)=0$.

The value $|f|$ of a flow f is the total flow out of s, or $\sum_v f(sv)$. A "flow cut" is a partition of $V(G)$ into two sets C and $\bar{C}=V(G)-C$ such that $s\in C, t\in\bar{C}$. The "capacity" of a cut is $cap(C\bar{C})=\sum_{v\in C,w\in\bar{C}} cap(vw)$. If $cap(C\bar{C})$ is the minimum over all possible cuts, then $C,\bar{C}$ is a minimum flow cut ("min-cut"). Summing the capacities of a min-cut solves the maximum flow problem.

We denote by $MIN\_CUT()$ any algorithm which produces minimum capacity cut sets $C,\bar{C}$ for any directed graph G.

Applied to our minimum attribute sacrifice problem, s and t intuitively correspond to an exclusive pair of attributes, and the resulting minimum capacity cut sets $C,\bar{C}$ are canonical visit sets. However, $MIN\_CUT()$ cannot immediately be used to solve the minimum sacrifice set problem since a min-cut minimizes the **capacity of the edges** spanning $C\bar{C}$. A minimum sacrifice set needs to minimize the **number of vertices** with defining edges spanning $C\bar{C}$. Using attribute optimization terminology, this is stated as follows:

Definition 3.13: Minimum Attribute Span

Given graph $IDS(X)$ with exclusive pair a,b and edges $c \rightarrow d, c \rightarrow_d d$ $(c \rightarrow_u d)$, a minimum attribute span is a division of the attributes of X into disjoint subsets S,M,T such that

(1) $a \in S \cup M,\ b \in T$.

(2) $w \in T,\ u \rightarrow_d w \Rightarrow u \in M \cup T$.

(3) $w \in T,\ w \rightarrow_u v \Rightarrow v \in T$.

(4) $|M|$ is minimal

Clearly, the set M corresponds to the notion of a minimum sacrifice set. What is now needed is a way to make attribute span problems "look" like network flow problems, whereby M can be found in polynomial time. We do this by transforming an $IDS(X)$ graph into a network flow graph G such that an edge spanning $C,\bar{C}$ produced from $MIN\_CUT(G)$ denotes a single sacrificed attribute. Figure 3.4 shows the transformation. Intuitively, given G constructed from $IDS(X)$ using these transformations, $MIN\_CUT(G)$ can only leave edges with cost 1 and 0 spanning $C,\bar{C}$. An edge of cost 1 represents a defining edge and thereby denotes a sacrificed attribute. An edge of cost 0 represents an induced edge. $MIN\_CUT()$ will clearly prefer cost 0 edges over cost 1 edges. Hence $MIN\_CUT()$ will find the cut set that minimizes the number sacrificed attributes. We must now prove that the transformation is correct.

w $$\begin{tikzcd} w1 \arrow[r, "1"] & w2 \end{tikzcd}$$

original attribute in $A(X)$ split vertex

vertex splitting
(a)

$$\begin{tikzcd} w \arrow[r, "d"] & v \end{tikzcd}$$

$$\begin{tikzcd} w1 \arrow[r, "1"] & w2 \arrow[r, bend left, "\infty"] & v1 \arrow[ll, bend left, "\infty"] \arrow[r, "1"] & v2 \end{tikzcd}$$

original edge in $IDS(X)$ transformation for flow algorithm

transformations to defining edges
(b)

$$\begin{tikzcd} w \arrow[r] & v \end{tikzcd}$$

$$\begin{tikzcd} w1 \arrow[r, "1"] & w2 \arrow[r, bend left, "0"] & v1 \arrow[ll, bend left, "0"] \arrow[r, "1"] & v2 \end{tikzcd}$$

original edge in $IDS(X)$ transformation for flow algorithm

transformations to induced edges
(c)

a,b $s=a1,\ t=b1$

original exclusive pair distinguished vertices for flow algorithm

transformed distinguished vertices
(d)

transformation from attribute sacrifice problem
to MIN-CUT/MAX FLOW problem
- Figure 3.4 -

Lemma 3.1:

Given G produced from instance $IDS(X)$ containing exclusive pair a,b according to the transformations of Figure 3.4, $MIN\_CUT()$ will produce a cut set $C,\bar{C}$ of finite capacity.

proof:

It suffices to show that a cut of finite capacity exists. Consider the following construction of $C,\bar{C}$ from $G=[V,E]$:

$$C=\{a_1\}\cup\{w_1 \mid w\rightarrow^{+}a\in IDS(X)\}$$

$$\bar{C}=V-C$$

Assume this construction has an infinite cost edge spanning $C\bar{C}$. According to Figure 3.4, there are only two possible ways for this to occur:

(1) $w_2\rightarrow_{\infty}v_1,\ w_2\in C,\ v_1\in\bar{C}$

(2) $w_1\rightarrow_{\infty}v_1,\ w_1\in C,\ v_1\in\bar{C}$

Case (1) is impossible by definition of C above. Case (2) is also impossible: $w_1\rightarrow_{\infty}v_1$ implies $v\rightarrow w\in IDS(X)$, but $w\rightarrow^{+}a\in IDS(X)$ means $v\rightarrow^{+}a$ also, so $v_1\in C$, a contradiction.

□ (lemma).

Theorem 3.1:

There exists a polynomial transformation [GaJ79] from the minimum attribute span problem to the maximum network flow problem.

proof:

Obviously, G can be created in polynomial time from $IDS(X)$ given the transformations of Figure 3.4. Consider executing $MIN\_CUT(G)$. After execution, the following must hold for each attribute $v\in A(X)$ split into v_1,v_2:

(1) $v_1 \in C, v_2 \in C$

(2) $v_1 \in C, v_2 \in \bar{C}$

(3) $v_1 \in \bar{C}, v_2 \in \bar{C}$

(4) $v_1 \in \bar{C}, v_2 \in C$

From this, the following sets can be produced:

$S = \{v \mid v_1, v_2$ satisfy possibility (1)$\}$

$M = \{v \mid v_1, v_2$ satisfy possibility (2)$\}$

$T = \{v \mid v_1, v_2$ satisfy possibilities (3) or (4)$\}$

Clearly S, M, T contain all the vertices of G, and can be produced in polynomial time from the output of *MIN_CUT*(). To see that S, M, T is a minimum attribute span for $IDS(X)$, consider each part of Definition 3.13 (minimum attribute span) in turn:

(1) (exclusive pair $a, b \in IDS(X) \Rightarrow a \in S \cup M, b \in T$): by Figure 3.4, exclusive pair $a, b \in IDS(X)$ implies distinguished $a_1, b_1 \in G$. This implies *MIN_CUT*() leaves $a_1 \in C$ and $b_1 \in \bar{C}$. This implies a_1, a_2 satisfy either of possibilities (1) or (2), and b_1, b_2 satisfy either of possibilities (3) or (4). This implies $a \in S \cup M, b \in T$.

(2) ($w \in T, u \rightarrow_d w \in G \Rightarrow u \in M \cup T$): assume $u \in S$. This implies *MIN_CUT*() left $u_2 \in C$. Likewise, $w \in T$ implies *MIN_CUT*() left $w_1 \in \bar{C}$. Then $u \rightarrow_d w \in G$ implies $u_2 \rightarrow_\infty w_1 \in \bar{G}$. This implies an edge of infinite cost spans $C\bar{C}$, contradictory to the finite capacity cut produced by *MIN_CUT*().

(3) ($w \in T, w \rightarrow v \Rightarrow v \in T$): assume $v \in S \cup M$. This implies *MIN_CUT*() left $v_1 \in C$. Likewise, $w \in T$ implies *MIN_CUT*() left $w_1 \in \bar{C}$. Then $w \rightarrow v \in G$ implies $v_1 \rightarrow_\infty w_1 \in \bar{G}$. This implies an edge of infinite cost spans $C\bar{C}$,

contradictory to the finite capacity cut produced by *MIN_CUT*().

(4) ($|M|$ is minimal): divide this proof into three parts:

(a) given a particular $IDS(X)$, G is uniquely defined. This and Lemma 3.1 imply that *MIN_CUT*() produces a finite capacity cut over G of minimum cost for given $IDS(X)$. Only edges $v_1 \rightarrow_1 v_2$ can contribute to this cost (all other edges have cost 0 or ∞). This implies that only split vertices satisfying possibility (2) ($v_1 \in C, v_2 \in \bar{C}$) contribute to the cost of the minimum cut. This implies that the set of vertices $v \in G$ such that v_1, v_2 satisfy possibility (2) is as small as possible.

(b) Let the number of vertices satisfying possibility (2) = $K = |M|$. Assume $\bar{S}, \bar{M}, \bar{T}$ is a minimum attribute span for $IDS(X)$ where $|\bar{M}| < K$. From $\bar{S}, \bar{M}, \bar{T}$ we can construct new $C', \bar{C}'$ as follows:

$$u \in \bar{S} => u_1, u_2 \in C$$
$$v \in \bar{T} => v_1, v_2 \in \bar{C}$$
$$w \in \bar{M} => w_1 \in C, w_2 \in \bar{C}$$

We show $C', \bar{C}'$ is a finite cut by considering all possible edges spanning the cut: spanning edges $u_2 \rightarrow v_1 \in G$ have cost 0; edge $u \rightarrow v \in IDS(X)$ is not defining since $u \in \bar{S}$. Spanning edges $u_1 \rightarrow_\infty v_1$ do not exist since then $v \rightarrow u$ would span from $\bar{T}$ to $\bar{S}$, which contradicts part (3) of Definition 3.13 (minimum attribute span). Edges $u_1, u_2 \rightarrow v_2, w_2$ do not exist by Figure 3.4. Likewise, edges $w_1 \rightarrow_\infty v_1$ do not exist, $w_1 \rightarrow v_2$ do not exist, and edges $w_1 \rightarrow w_2$ have cost 1. Hence $C, \bar{C}$ is a cut of finite capacity.

(c) The number of vertices satisfying (2)=$|\bar{M}| < K$, which contradicts (a) and (b). Hence no $\bar{S}, \bar{M}, \bar{T}$ exists which is a legal network span where

$|\bar{M}|<K$. Since $|M|=K$, this implies the original S,M,T is a legal minimum attribute span.

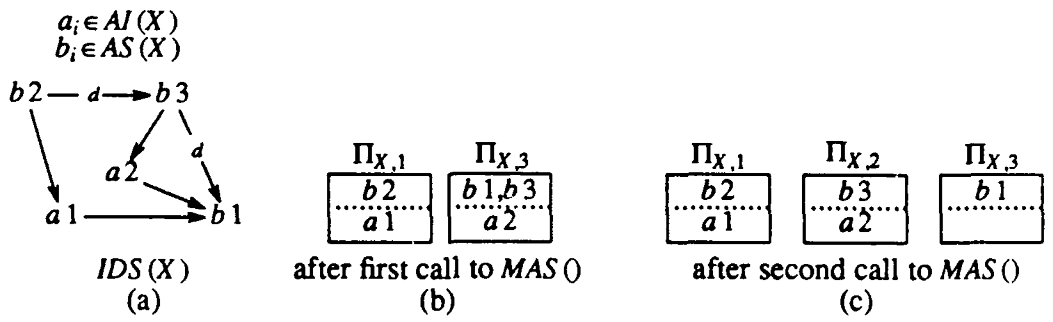
Q.E.D.□ (theorem)

3.5.2. Partitioning algorithm

Let us denote by $MAS(IDS,s,t)$ the minimum attribute span procedure outlined in the previous theorem, and let $IDS(X)$ contain a single exclusive pair a,b. Assuming $MAS()$ produces as output the sets S,M,T, canonical visit sets can be produced:

$$\Pi_{X,1}=S \cup M,\ \Pi_{X,2}=T$$

Of course, the single exclusive pair a,b is used as s,t and satisfies part (1) of Property 3.2 (visit sets). Part (3) of Definition 3.13 (minimum attribute span) ensures that no edges exist from $\Pi_{X,2}$ to $\Pi_{X,1}$, which satisfies part (2) of Property 3.2 (visit sets). Finally, parts (2) and (4) of Definition 3.13 ensure that the number of sacrificed attributes is minimal. Canonical visit sets are therefore optimal in the case where one minimal exclusive pair exists in $IDS(X)$.



application of $MAS()$ to $IDS(X)$ having multiple instances of exclusive pairs
- Figure 3.5 -

Producing an algorithm for the general case of partitioning (when two or more minimal exclusive pairs are present) requires extra work. Consider the $IDS(X)$ graph of Figure 3.5a. Here, an algorithm to produce visit sets for $IDS(X)$ can call $MAS()$ twice, successively using the two minimal exclusive pairs a_1,b_1, a_2,b_1 as distinguished s,t. Suppose the first call is $MAS(IDS(X),a_1,b_1)$. This results in the canonical visit sets of Figure 3.5b. These visit sets are not yet legal, however, since the exclusive pair a_2,b_1 is contained within $\Pi_{X,2}$, violating part (1) of Property 3.2 (visit sets). $MAS()$ must be called again, this time to split the left-over exclusive pair. Since calling $MAS()$ with the entire graph $IDS(X)$ as its first argument is redundant, the subset $IDS'(X)$ containing only those $v \in \Pi_{X,2}$ is used. The call $MAS(IDS'(X),a_2,b_1)$ then produces S,M,T such that two new canonical visit sets are added to $\Pi_{X,1}$, yielding Figure 3.5c. There are no more exclusive pairs to split apart, and so no further calls to $MAS()$ are necessary.

So far, visit sets produced via repeated calls to $MAS()$ are not always optimal because this creates more visit sets than are actually required. For example, there is no reason why $\Pi_{X,1}$ of Figure 3.5c cannot be merged with $\Pi_{X,2}$, yielding optimal visit sets.

Definition 3.14: (compatible visit sets):

two legal visit sets $\Pi_{X,i},\Pi_{X,j}$ are "compatible" if their union does not contain any exclusive pairs (note: compatibility is not transitive).

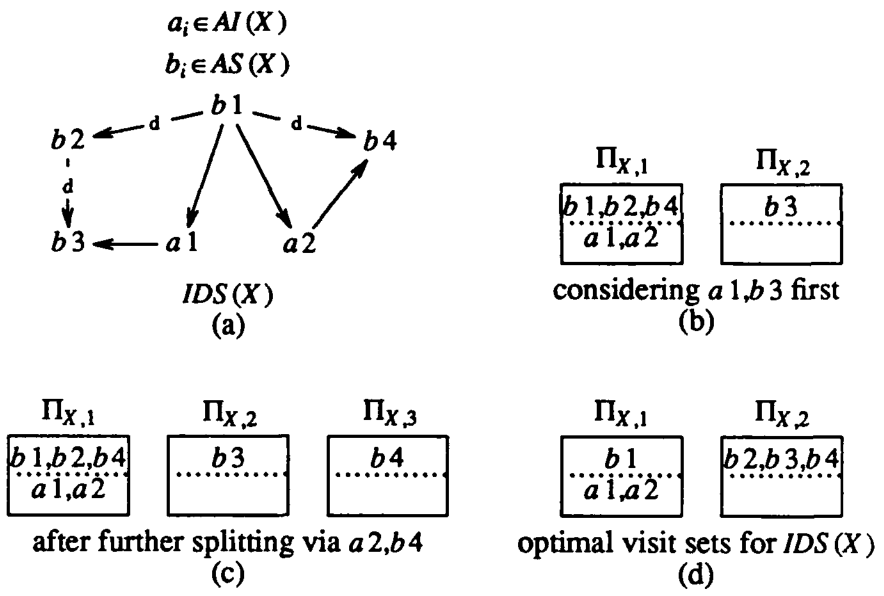
The new problem arises of how to group together compatible canonical visit sets for a symbol. It is best to group them such that the number of total visits to a symbol is minimized. With this in mind, it is illustrative to consider the way the OAG partitioning strategy assigns attributes to partitions. When dividing attributes of a symbol into partitions (not visit sets), inherited and synthesized attributes are "incompatible" (just as visit sets are). Suppose some $IDS(X)$ has a path of alternating inherited and synthesized attributes like so:

$$b_1 \rightarrow a_1 \rightarrow \ldots \rightarrow b_n \rightarrow a_n, \text{ where } b_i \in AI(X), a_i \in AS(X).$$

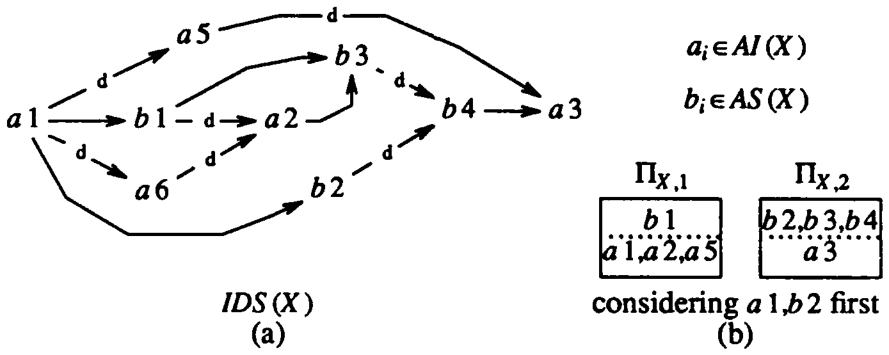
It is obvious that **at least** $2 \times n$ partitions are needed to accommodate this $IDS(X)$. If this is the longest alternating path in $IDS(X)$, then we speculate from the description of building OAG partitions in [Kas80] that **no more than** $2 \times n$ attributes will be needed to accommodate this graph (no proof will be given here to substantiate this claim[7]). The new problem of grouping canonical visit sets looks very much like the problem of grouping attributes into partitions; each visit set looks just like a ''big fat'' attribute. Dependency relationships exist between visit sets just as for attributes, and certain visit sets are incompatible with others, just as for attributes. It may be possible, in this sense, to adapt the OAG partitioning strategy to the visit set merging problem; even if the OAG strategy does not actually produce the minimum number of partitions, it may still do a ''good enough'' job for the purposes of visit set merging.

Although repeated application of $MAS()$ with merging will produce acceptable results in any case, there are some cases where such application fails to be optimal (even if we have optimal merging). Figure 3.6 contains one such example. Even though b_1 is a forced span, if $MAS()$ is given a_1, b_3 first as distinguished s, t, then there is nothing to prevent the sacrifice of b_2, yielding Figure 3.6b. Further splitting of $\Pi_{X,1}$ sacrifices b_1 as well, yielding Figure 3.6c. Afterwards, even though $\Pi_{X,2}, \Pi_{X,3}$ can be merged, both b_2, b_1 will be spanning attributes. However, only b_1 spans in the optimal visit sets of Figure 3.6c. Fortunately, $MAS()$ can be used to produce optimal visit sets even for this case, but some preprocessing of $IDS(X)$ is required:

[7]We speculate that the dependency relationships between attributes keeps the incompatible attribute grouping problem from reducing to the **general** incompatible object grouping problem, which is known to be NP-complete.



application of *MAS*() to *IDS*(*X*) having multiple instances of exclusive pairs
- Figure 3.6 -



effectiveness of *MAS*() depends on which of the exclusive pairs is considered first
- Figure 3.7 -

```
foreach $v \in IDS(X)$
  if Forced_Span($v$)
    foreach $v \rightarrow_d w \in IDS(X)$
      replace $v \rightarrow_d w$ with $v \rightarrow w$
```

This will ensure that *MAS*() will not try to save nodes which will have to be sacrificed anyway. If $IDS(X)$ of Figure 3.6a is preprocessed this way, iterative execution of *MAS*() produces the optimal visit sets of Figure 3.6d, where only b_1 is spanning.

An example of a more sinister $IDS(X)$ graph is given in Figure 3.7. If exclusive pair a_1, b_2 is considered first, there is nothing to stop *MAS*() from producing S, M, T which result in the incomplete visit sets of Figure 3.7b. This sacrifices a_5. $\Pi_{X,1}$ must again be split, which will sacrifice either of a_1, a_6. The resulting visit sets will have two spanning attributes, and hence are not optimal for $IDS(X)$. Optimal visit sets have the single spanning attribute a_1. Removing forced spans from $IDS(X)$ does not help in this situation. Instead, other preprocessing is required. We note that a_1 is a contingent attribute in two distinct cases: defining path $a_1 \rightarrow_d^+ a_2$ spanning $a_1 \rightarrow b_1 \rightarrow a_2$, and defining path $a_1 \rightarrow_d^+ a_3$ spanning $a_1 \rightarrow b_2 \rightarrow^+ a_3$. On the other hand, a_5 is contingent in only one case. Sacrificing a_1 in the context of one exclusive pair just might save the sacrifice of a different attribute in another context. Since a "tie" occurs between two the attributes, such that sacrificing either would produce a minimal cut set, *MAS*() should choose a_1, which "saves" attributes later.

Definition 3.15: (utility):

The "utility" of an attribute $u(X.a)$ is computed as follows:

```
$u(X.a) := 0$
foreach minimal exclusive pair $v, w \in IDS(X)$
  if $a$ is contingent in the context of $v, w$
    $u(X.a) := u(X.a) + 1$
```

If *MAS*() were able to take advantage of the utility of $IDS(X)$ then, for Figure 3.7, it would sacrifice a_1 instead of a_5 when the "choice" between them arises. Optimal visit sets with single spanning a_1 would then be produced from *MAS*().

Of course, this utility information should only be used in deciding which of two contingent attributes should be sacrificed. In no way should it ever allow two attributes to be sacrificed when one would suffice.

Definition 3.16: (Maximum Utility Attribute Span):

Given $IDS(X)$, a maximum utility attribute span S, M, T for $IDS(X)$ is a minimum attribute span for $IDS(X)$, such that no other minimum attribute span $\bar{S}, \bar{M}, \bar{T}$ exists for $IDS(X)$ where $\sum_{\bar{m} \in \bar{M}} u(\bar{m}) > \sum_{m \in M} u(m)$.

U_X = highest utility in $IDS(X)$
$|A|$ is the size of $A(X)$
$u(w)$ is the utility of attribute w

$$w \qquad\qquad w1 \xrightarrow{\quad |A|U_X+1-u(w) \quad} w2$$

original attribute in $A(X)$ split vertex

vertex splitting for Maximum Utility Span
(all other transformations same as Figure 3.4)
(a)

cut sets created by *Min_Cut*() are $C, \bar{C}$

$w_1 \in C, w_2 \in \underline{C}$	=>	$w \in S$
$w_1 \in \underline{C}, w_2 \in \overline{C}$	=>	$w \in M$
$w_1 \in \underline{C}, w_2 \in \underline{C}$	=>	$w \in T$
$w_1 \in \overline{C}, w_2 \in \overline{C}$	=>	$w \in T$
possibilities for split vertices after execution of *Min_Cut*()		proper placement of original vertices

creation of S, M, T from output of *Min_Cut*()
(b)

transformation from Maximum Utility Spanning Network problem
to Minimum Span Network problem
- Figure 3.8 -

A way must now be devised to get *MIN_CUT*() to take advantage of utility counts for attributes, while adhering to the constraints of the minimum attribute span problem. Unfortunately, the maximum network flow problem does not allow for multiple weights on the edges when choosing an minimal cut set to output. It also does not produce all minimal cut sets, from which the one with maximum utility could be chosen. Some form of single edge weights must be used to achieve the desired results. We give appropriate edge weights in Figure 3.8a. Intuitively, the edge weight reflects the very high cost of sacrificing an attribute, while the attribute's utility metric reduces that cost by a small amount. The trick is to make the cost of sacrificing a single attribute so high that no matter what the utility of two different attributes, sacrificing them both would cost more. Of course, this must be shown to hold in all cases.

Lemma 3.2

Given $W \subset A(X)$, define $cost(W) = \sum cap(w_1 w_2)$ where w_1, w_2 are produced from $w \in W$ via transformations in Figure 3.8a. We show that $M_1, M_2 \subset A(X)$ and $|M_1| < |M_2|$ implies $cost(M_1) < cost(M_2)$.

proof

by definition:

$$cost(M) = \sum_{m_1, m_2 \in M} cap(m_1 m_2)$$

We pursue this further. According to Figure 3.8:

$$cost(M) = U_X |A| |M| + |M| - \sum_{m \in M} u(m) \Rightarrow$$

$$cost(M) \leq U_X |A| |M| + |M| \Rightarrow$$

$$cost(M) \leq U_X |A| |M| + |M| + U_X |A| - U_X |A| \Rightarrow$$

$$cost(M) \leq U_X |A| |M| + |M| + U_X |A| - U_X (|M| + 1) \Rightarrow$$

$$cost(M) \leq U_X |A| |M| + |M| + U_X |A| - U_X |M| - U_X \Rightarrow$$

$$cost(M) < U_X|A||M|+|M|+U_X|A|-U_X|M|-U_X+1 \Rightarrow$$

$$cost(M) < (|M|+1)(U_X|A|-U_X+1) \Rightarrow$$

$$cost(M) < |\overline{M}|(U_X|A|-U_X+1) \Rightarrow$$

$$cost(M) < U_X|A||\overline{M}|+|\overline{M}|-U_X|\overline{M}| \Rightarrow$$

$$cost(M) < U_X|A||\overline{M}|+|\overline{M}|-\sum_{\overline{m}\in\overline{M}} u(\overline{m}) \Rightarrow$$

$$cost(M) < cost(\overline{M})$$

□ (lemma)

Theorem 3.2:

Figure 3.8 provides a polynomial transformation [GaJ79] from the maximum utility attribute span problem to the maximum network flow problem.

proof

Certainly the graph G can be produced from $IDS(X)$ according to the transformations of Figure 3.8a in polynomial time. Also, S, M, T can be produced in polynomial time (as described in Figure 3.8b) from output $C, \overline{C}$ of $MIN\_CUT()$. So the transformation is indeed polynomial. We must now show that the transformation is correct. The following must be shown:

(a) S, M, T is a minimum attribute span for G.

(b) there is no other minimum attribute span $\overline{S}, \overline{M}, \overline{T}$

where $\sum_{\overline{m}\in\overline{M}} u(\overline{m}) > \sum_{m\in M} u(m)$.

To show (a), it must be proven that a finite cut set for G exists, and that S, M, T obey the four properties of a minimum attribute span (MAS). The proof of finiteness, and the first three properties of MAS, are trivial modifications of the proof that the transformations in Figure 3.4 solve the minimum attribute span problem, and will not be given here. However, the fact that $|M|$ is minimal must still be shown. Assume network span S',M',T' exists, and that $|M|>|M'|$. M was produced from the output of $MIN\_CUT(G)$, where edge $m_1 \rightarrow m_2$ spanned the cut $C,\bar{C}$. Only these edges contribute to $cap(C\bar{C})$. Let $cost(M)=\sum_{m_1,m_2 \in M} cap(m_1 m_2)$, then $cost(M)=cap(C\bar{C})$. But Lemma 3.2 shows $cost(M')<cost(M)$ whenever $|M'|<|M|$. Since a legal cut set $C',\bar{C}'$ could therefore be produced from S',M',T' with $cap(C'\bar{C}')=cost(M')<cost(M)$, S',M',T' must not exist.

To show (b), suppose such $\bar{S},\bar{M},\bar{T}$ exists. But,

$$\sum_{\bar{m}\in\bar{M}} u(\bar{m}) > \sum_{m\in M} u(m) \quad =>$$

$$|M|+|M|\,|U_X|\,|A| - \sum_{\bar{m}\in\bar{M}} u(\bar{m}) < |M|+|M|\,|U_X|\,|A| - \sum_{m\in M} u(m) \quad =>$$

$$|\bar{M}|+|\bar{M}|\,|U_X|\,|A| - \sum_{\bar{m}\in\bar{M}} u(\bar{m}) < |M|+|M|\,|U_X|\,|A| - \sum_{m\in M} u(m) \quad =>$$

$$\sum_{\bar{m}\in\bar{M}} cap(\bar{m}_1\bar{m}_2) < \sum_{m\in M} cap(m_1 m_2) \quad =>$$

$$cost(\bar{M})<cost(M)$$

This means that $MIN\_CUT()$ could have produced a cut set with capacity no more than $cost(\bar{M})$. This contradicts optimality of $MIN\_CUT()$. Hence $\bar{S},\bar{M},\bar{T}$ must not exist.

□ (theorem)

It has now been established that the maximum network flow problem can be used to produce "good" partitions of attributes, even where multiple instances of minimal exclusive pairs are involved. The algorithm of Figure 3.9 applies the techniques described above to produce such visit sets from an original $IDS(X)$ graph. Note that the algorithm considers only minimal exclusive pairs. The algorithm halts since the number of minimal exclusive pairs is finite, and executing MUS over

```
/* On Entry:
        Given: graph $IDS(X)$, procedure $MUS()$ described above.
        Given: procedure $merge(\Pi_X)$ which recombines
               legal, compatible visit sets.
        Given: procedure $Forced\_Span(G)$ replaces all edges $a \rightarrow_d b \in IDS(X)$
               by $a \rightarrow b$ when $a$ is a forced span.
        Given: procedure $count(G)$ which performs utility analysis of $IDS(X)$.
  On Exit:
        $\Pi_{X,i}$ are canonical visit sets using the Near Partitioning strategy
*/
$NAP(X)$ {
    $G_0 := Forced\_Span(IDS(X))$;
    while $\exists a \rightarrow_u b \in E[G_i]$ such that $a,b$ is a minimal exclusive pair do {
        $count(G_i, utility)$; /*initialize utility */
        $S, M, T := MUS(G_i, a, b)$; /*$utility[v_i]$ used*/
        /*create canonical partitions*/
        $V[G_j] := T$;
        $E[G_j] := w \rightarrow_u v$ such that
                    $w, v \in V[G_j]$ and $w \rightarrow_u v \in E[G_i]$;
        $V[G_k] := S \cup M$;
        $E[G_k] := v \rightarrow_u w$ such that $v \in S$, $w \in M$, $v \rightarrow_u w \in E[G_i]$
              AND /*sacrifice $m \in M$ */
              $w \rightarrow v$ such that $v \in S$, $w \in M$, $w \rightarrow_u v \in E[G_i]$;
        $delete(G_i)$;
    } /*while*/
    foreach $G_i$ still remaining do
        $\Pi_{X,i} := V[G_i]$;
    /*recombine canonical partitions where possible*/
    $merge(\Pi_{X,i})$;
} /*NAP */
```

procedure which produces visit sets for given $IDS(X)$
via the Near as Possible (NAP) Partitioning Strategy

Figure 3.9

an exclusive pair guarantees they will be thereafter placed in different $V[G_i]$. The absence of minimal exclusive pairs implies the absence of exclusive pairs, so the resulting $\Pi_{X,i}$ constitute legal visit sets. Finally, note that the algorithm produces visit sets such that the set of spanning attributes is exactly equal to the union of all sacrifice sets M. Furthermore, *merge*() reduces the number of visits given this sacrifice set.

This concludes the discussion of the optimization of attributes via better partitioning strategies. Attention now focuses on the only other phase of creating visit sequences over which we have control.

3.6. Impact of topological sort of *DP*(*p*) on storage optimization

As [FaY86] points out, the characteristics of "good" sets of visit sequences are (in order of importance):

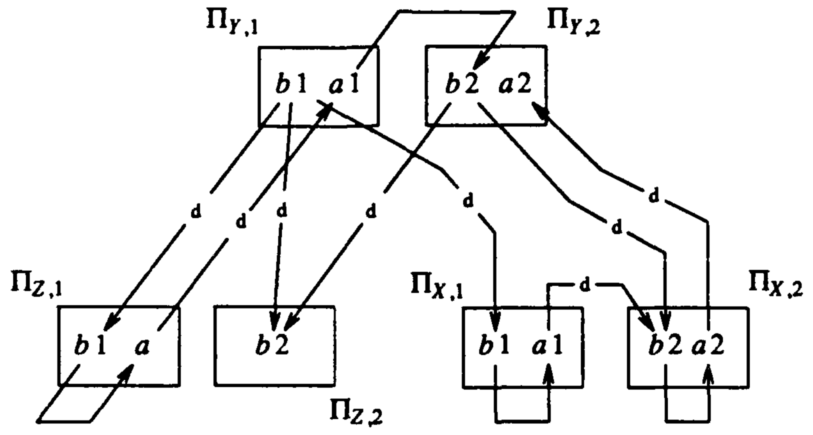
(1) They minimize the number of multi-visit attributes.

(2) For single visit attributes, they maximize the number implemented as global variables.

(3) For those single visit attributes which are not implemented as global variables, they maximize the number implemented as stacks pushed and popped in different productions (note that we already stack all synthesized attributes this way).

(4) For stackable attributes, they maximize the number of instances of *CLOBBER* stack operations.

Unfortunately, creating even the best possible visit sets does not guarantee that visit sequences generated from them will have these favorable characteristics. As described earlier, once visit sets have been established for all the grammar symbols of a simple multi-visit AG, the next step is use them to create the graph $DP(p)$ for the grammar productions, and finally to perform a topological sort on each $DP(p)$. It is the variety of possible methods for performing this topological sort that keeps visit sequences from being unique, and in general one method will be better than another for each of the optimizations (1)-(4) above. We substantiate this claim with examples of where each optimization can be foiled:



bad v: $\downarrow_1 Z, \downarrow_1 X, \uparrow_1, \downarrow_2 Z, \downarrow_2 X, \uparrow_2$
good v: $\downarrow_1 Z, \uparrow_1, \downarrow_2 Z, \downarrow_1 X, \downarrow_2 X, \uparrow_2$

visit scheduling impacts number of multi-visit attributes
- Figure 3.10 -

$p: X ::= \cdots X \cdots$
  $r_{p,1}: X[2].b := f(\cdots X[1].b \cdots);$
  $r_{p,2}: X[2].d := g(\cdots X[1].b \cdots);$
  $\vdots$

bad v: $\cdots \rightarrow X[2].b, \rightarrow X[2].d, \downarrow_1 X[2], \cdots$
good v: $\cdots \rightarrow X[2].d, \rightarrow X[2].b, \downarrow_1 X[2], \cdots$

order of evaluation of attributes affects number of global variables given a fixed visit schedule
- Figure 3.11 -

$p : X ::= X\ Z$
  $r_{p,1}$: $Z.b := f(\cdots X[1].b \cdots);$
  $r_{p,2}$: $X[2].b := g(\cdots X[1].b \cdots);$
  ⋮

bad v: $\cdots \rightarrow X[2].b, \downarrow_1 X[2], \rightarrow Z.b, \downarrow_1 Z, \cdots$
good v: $\cdots \rightarrow X[2].b, \rightarrow Z.b, \downarrow_1 X[2], \downarrow_1 Z, \cdots$

order of evaluation of attributes affects number
of stacks pushed and popped in different productions
given a fixed visit schedule

- Figure 3.12 -

(1) To see how visit sequences affect the number of multi-visit attributes, consider Figure 3.10. The visit schedule of the "bad" visit sequence places $\downarrow_1 X$ and $\downarrow_2 X$ on either side of $\uparrow_1$. This causes $X.a_1$ to be multi-visit (recall that the parent attribute $Y.b_1$ is not regarded as multi-visit even though it defines attributes in both visits to the production). This visit schedule was created via a greedy strategy, which inserts $\rightarrow X.a_k$, $\downarrow_i X$ into a visit sequence **as soon** as all $X.a_k \in \Pi^{I}_{X,i}$ can be evaluated. $\downarrow_1 X$ is inserted before the first *LEAVE* simply because the value of $Y.b_1$ is available then. But $\downarrow_2 X$ cannot be inserted along with it because the value of $Y.a_1$ is not available to evaluate $X.b_2$. In the "good" visit sequence, $\downarrow_1 X$ is inserted "nearer" to $\downarrow_2 X$. We use a lazy strategy in this case, which calls for insertion of $\downarrow_i X$ only when it **cannot** be inserted later. This suffices to keep $X.a_1$ from being multi-visit for the situation given. However, it is easy to imagine that the lazy strategy will not work optimally in all situations either, leaving the way clear for a better strategy.

(2) To see how a visit sequence affects the number of global variables, consider Figure 3.11. In the bad visit sequence, the lifetime of $X[1].b$ overlaps the lifetime of $X[2].b$, so it cannot be implemented as a global variable. This situa-

tion occurs solely because of the order of evaluation of $X.b, X.d$. A smarter strategy yields the good visit sequence, which (by itself) allows $X.b$ to be a global variable.

(3) To see how a visit sequence affects the number of stacks pushed from below, consider Figure 3.12. According to the bad visit sequence, $X.b$ must be implemented as a stack pushed and popped from above. This is because the value of $X[2].b$ is still needed after the visit to X, and so the *POP* for $X.a$ cannot occur during $\downarrow_i X$. Of course, care must be taken so that $Z.b$ does not acquire a less desirable storage class in the good visit sequence because of its placement before $\downarrow_i X$. For example, if $Z.b$ can be implemented as a global variable, and X derives Z (as it does), then the movement of $\rightarrow Z.b$ before the visit possibly extends its storage implementation to stack. However, if $Z.a$ is already a stack, moving its evaluation in this way cannot change its class from "stack from below" to "stack from above." The consequence of not performing the move when $Z.a$ is a stack is that $X.b$ would then be stacked from above in every production. So the move is a gain in the sense that the "stack-time" of one attribute is shortened for every production, at the cost of lengthening the "stack-time" of another attribute for a single production.

(4) To see how a visit sequence affects the number of *CLOBBER*s used, Figure 3.11 can again be considered. If $X.d$ is implemented as a stack popped from below, then the good visit sequence will allow $\rightarrow X[2].b$ to be a *CLOBBER*, instead of the *PUSH*/*POP* pair required by the bad visit sequence.

Given these examples, it is evident that some intelligence is required not only to decide when to schedule visits, but also to decide in what order to evaluate attributes given a visit schedule.

3.6.1. Producing a visit sequence

Creating a visit sequence is a two phase process. The first phase dictates during which visit to a **production** the visits to its **children** are to be scheduled. It is this phase that determines which attributes will be multi-visit. The second phase produces a total ordering of the *VISIT* s and *EVAL* s that must occur between successive visits to the production. This ordering must be consistent with the dependencies in the *IDS* graph for each symbol and also with the dependencies of the *DP* graphs. Although not as important as keeping attributes from being multi-visit, this stage does determine to what extent the lesser optimizations (2-4 described above) occur.

Each $DP(p)$ graph is very similar to an $IDS(X)$ graph. They also contain defining edges and induced edges, and instances of forced and contingent spans. Consider the example AG production and *DP* graph given in Figure 3.13 (based on [WaG83, p. 200]). The inherited attributes of the parent ($\Pi^I_{Y,i}$) and the synthesized attributes of the children ($\Pi^S_{X,i}$,$\Pi^S_{Z,i}$) are all evaluated by the TWA outside the context of p. Furthermore, all the attributes of any one such partition must be evaluated during the same *LEAVE* or *VISIT*, so each $\Pi^I_{Y,i}$, $\Pi^S_{X,i}$, and $\Pi^S_{X,i}$ corresponds to a single vertex in $DP(p)$.

The similarity of the graphs used in visit set production and visit sequence production (*IDS* vs *DP*) immediately suggests that a maximum network flow solution can also be used for the latter.

3.6.2. Visit scheduling

As in the partitioning problem, it is desirous to produce an ordering of the visits to X,Z for production $p: Y ::= Z\ X$ such that the number of defining arrows spanning the visits to Y (the *LEAVE* s) is minimal. It could be argued that

$p: Y ::= X\ Z\ X$

r_1: $X[1].e := Y.e$;

r_2: $X[2].e := Y.e$;

r_3: $Y.pr := f\,(X[1].pr, X[2].pr)$;

r_4: $Z.m := Y.pr$;

r_5: $X[1].po := Y.pr$;

r_6: $X[2].po := Y.pr$;

(a) - production and semantic rules

$\Pi^I_{Y,1} = \{e\}$ $\qquad$ $\Pi^I_{X,1} = \{e\}$

$\Pi^S_{Y,1} = \{pr\}$ $\qquad$ $\Pi^S_{X,1} = \{pr\}$

$\Pi^I_{Y,2} = \{po\}$ $\qquad$ $\Pi^I_{X,1} = \{po\}$

$\Pi^S_{Y,2} = \varnothing$ $\qquad$ $\Pi^S_{X,2} = \varnothing$

$\Pi^I_{Z,1} = \{m\}$

$\Pi^S_{Z,1} = \{op\}$

(b) - partitions

$$\begin{tikzcd}
 & \Pi^I_{Y,1} \arrow[dl, "d"'] \arrow[dr, "d"] \arrow[dddd, bend left=40] \arrow[dddddd, bend right=40] & \\
X[1].e \arrow[dd] \arrow[dddd, bend right=40] & \Pi^S_{Z,1} & X[2].e \arrow[dd] \arrow[dddd, bend left=40] \\
 & Z.m \arrow[u] & \\
\Pi^S_{X[1],1} \arrow[ur] \arrow[dr, "d"] \arrow[dd] \arrow[dddd, bend right=50] & & \Pi^S_{X[2],1} \arrow[ul] \arrow[dl, "d"'] \arrow[dd] \arrow[dddd, bend left=50] \\
 & Y.pr \arrow[uu, "d"] \arrow[dl, "d"'] \arrow[dr, "d"] \arrow[dd] & \\
X[1].po \arrow[dd] & & X[2].po \arrow[dd] \\
 & \Pi^I_{Y,2} & \\
\Pi^S_{X[1],2} & & \Pi^S_{X[2],2}
\end{tikzcd}$$

graph $DP(p)$

(c)

partitions and DP graph for attribute grammar production

- Figure 3.13 -

since the production context for p does not decide the multi-visit property for Y's attributes, then they should be considered unimportant when processing $DP(p)$. However, this might cause otherwise non-spanning attributes of the parent to wind up being multi-visit. Consider deleting the edge $Y.b_1 \rightarrow_d Z.b_2$ in Figure 3.10. This means the lifetime of $Y.b_1$ is no longer **required** to span the two visits to Y. Now, if the visit sequence for some other production with child symbol Y inserts $\uparrow_i$ between $\downarrow_1 Y, \downarrow_2 Y$, then the lazy insertion of $\downarrow_1 X$ during the second visit to Y causes $Y.b_1$ to be multi-visit. On the other hand, the lack of $\uparrow_i$ between $\downarrow_1 Y, \downarrow_2 Y$ in any production where Y is a child symbol would mean that $Y.b_1$ is not multi-visit, whereas splitting $\downarrow_1 X, \downarrow_2 X$ in p **guarantees** that $X.a_1$ is multi-visit, so at least in the situation of Figure 3.10 it is better to sacrifice $Y.b_1$. But suppose several of Y's attributes, with lifetimes similar to $Y.b_1$, were used to define $X.b_1$. Any policy of recklessly sacrificing them all for the single $X.a_1$ could be disastrous (particularly so, if it turns out that $X.a_1$ is a forced span in some other $DP(q)$, as described in the next paragraph). Parent attributes should be sacrificed before children attributes only when an otherwise even tradeoff between sacrificing either group exists.

More important than the notion of whether an attribute belongs to a child or a parent symbol is the notion of its utility throughout all the productions. Certainly, if an attribute is a forced span in one $DP(p)$, any defining arrows it has emanating from it should be replaced by induced edges in all other $DP(q)$. Also, should one attribute be involved in many more DP graphs as a contingent span than some other attribute, the one with a high span count should be sacrificed before the other, all else being equal. With these properties in mind, algorithms to preprocess DP graphs can be developed. From preprocessed DP graphs, better visit sequences can be produced.

The first thing we must preprocess in the *DP* graphs are the child vertices $\Pi^{S}_{X,i}$. It is clear that ''sacrificing'' one of these nodes (making it span a *LEAVE*) can have a cost of more than one attribute. In fact, it could be the case that every attribute in one of these partitions winds up spanning a *LEAVE* whenever one of them did. To prevent sacrificing them all at a cost of one, the transformation in Figure 3.14a to the *DP* graph is proposed, which exposes the defining contributions of each attribute in some Π^{S} of a child symbol. The introduction of the ''left-locking'' and ''right-locking'' vertices $L_{X,i}, R_{X,i}$ ensures that all the attributes of $\Pi^{S}_{X,i}$ will be evaluated during the same visit. Furthermore, the exposure of each individual attribute's defining edges will keep them from being sacrificed without regard to cost.

original vertex of $DP: \Pi^{S}_{X,i}$
where $X.d_1 \cdots X.d_k \in \Pi^{S}_{X,i}$

$$\begin{tikzcd}
 & X.d_1 \arrow[dr, "\infty"] & \\
L_{X,i} \arrow[ur, "\infty"] \arrow[r, "\infty"] \arrow[dr, "\infty"'] & X.d_2 \arrow[r, "\infty"] & R_{X,i} \\
 & \vdots & \\
 & X.d_k \arrow[uur, "\infty"'] &
\end{tikzcd}$$

exposing synthesized attribute contribution
of the child symbols in *DP*, without
violating partitioning conditions
(a)

$$\begin{tikzcd}
w1 \arrow[r, "|A|^2 U_X - |A| u(w) + (0,1)"] & w2
\end{tikzcd}$$

vertex splitting for optimizing over
sacrificed attributes, utility, and
parent or child symbol, for attribute w
(b)

preventing uncontrolled sacrifice of synthesized
child symbol attributes
- Figure 3.14 -

The second preprocessing step of *DP* graphs is to transform them as *IDS* graphs were transformed: to get *MIN_CUT*() to minimize the number of attributes sacrificed. Since it is still desirable to sacrifice vertices with high contingent span counts first, and since parent attributes are better sacrifices than child attributes, the transformations of Figure 3.14b are proposed. The splitting of vertices into two is done exactly as in Figure 3.8. The edge weights cause first the sacrifice of the least number of vertices and then, among such "least number" solutions, the one with (first) highest contingent span count and (second) lowest child attribute count are chosen. The proof that this is a correct polynimial time transformation to the maximum network flow problem is a straightforward modification of the proof showing Figure 3.8 is a correct transformation, and so will be omitted here.

These ideas lead to the algorithm of Figure 3.15 which solves the visit scheduling problem. Since the number of visits to any production p is finite, the algorithm halts. Every visit to a child symbol is assigned to some $vs[p,i]$ since, during the last iteration over i, the call to *MUS* in the inner loop places every attribute not previously assigned into $S \cup M$. The order of the productions analyzed will in general affect the outcome of the vs array, since those attributes sacrificed while processing early *DP* graphs are noted as such when analyzing later *DP* graphs. It should be noted here that just because an attribute was sacrificed by *MUS* in the procedure does not necessarily mean that it must remain a tree based attribute. Suppose an attribute $Y.b$'s lifetime spans a *LEAVE* from some production context p where X is the parent symbol, but X has "capped" visits, meaning $\downarrow_i X, \downarrow_{i+1} X$ are not separated by $\uparrow_j$ in any production context where X is a child symbol. In this case, $Y.b$ is still implementable globally.

```
/* On Entry:
        $DP(p)$ is given for each $p$
        $count(DP(p))$ performs utility analysis over $DP(p)$
        $MUS(DP(p),s,t)$ is a maximum utility minimum attribute span algorithm
        $reduce\_count(M)$ reduces the utility of any attribute on a
                contingent span path with some $m \in M$ by one.
        $transform(DP(p))$ performs the transformations of Figure 3.14.
        $Forced\_Span(AG)$ replaces all $w \rightarrow_d v$ in every $DP(p)$
                with $w \rightarrow v$ if $w$ is forced span in any $DP(q)$.
  On Exit:
        $vs[p,i]$, contains those $\downarrow_k$ to child symbols which must
                occur during the $i^{th}$ visit to the production $p$.
*/
            Visit_Schedule(AG) {
              Forced_Span(AG);
              count(AG);
              foreach production $p$ {
                $DP'(p) := transform(DP(p))$;
                foreach $i := 1 \cdots MAX$(visit number for $p$)$-1$ {
                  /*use LEAVEs from $p$ as distinguished $s,t$*/
                  $MUS(DP'(p),\Pi_{X,i},\Pi_{X,i+1})$;
                  $vs[p,i] := S \cup M$;
                }
                reduce_count(AG,M);
              }
            }
```

visit scheduling algorithm

- Figure 3.15 -

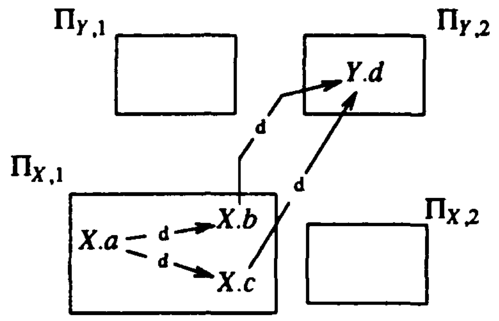
3.7. Postprocessing visit sequences

Once the visit sequences for every production have been created, extra processing might make them even better. This stems from the fact that a *DP* graph cannot be created without the **a-priori** existence of partitions for every symbol. On the other hand, we cannot really determine the effectiveness of partitions until the *DP*(*p*) graphs are built.

In order to get around this circular dependence, some postprocessing of visit sequences will give good results. Once visit sequences are built, certain inappropriate decisions that were made at the partitioning level are exposed, which can be fixed without having to go back to the partitioning phase. At this point, certain "shuffle" operations can be performed on the visit sequences which do not violate their static properties.

To see this problem more clearly, consider the attribute dependencies depicted in Figure 3.16. The partitions for X reflect the fact that, according to the strict dependencies within a symbol, $X.a, X.b, X.c$ should all be contained in the same visit set. However, when augmented by the dependencies of p, $X.b, X.c$ turn out to be multi-visit attributes. In general, it cannot be known at visit set creation time that saving $X.a$ will cause the eventual sacrifice of $X.b, X.c$, because only the placement of $Y.d$ into $\Pi_{Y,2}$ and the creation of v for p causes this. Hence, to create the best possible visit sets, all the visit sets must already be known. It is possible to **iteratively** construct visit sets, building new ones once the visit sequences are known, but at substantial recomputation cost. Fortunately, the following simple alternative can produce good results without recomputation.

Consider the partitioning strategy of Figure 3.16, which sacrifices the two attributes $X.b, X.c$ for the sake of $X.a$. An algorithm can inspect this production and easily determine that $X.b, X.c$ are two attributes which are sacrificed for the sake of the single attribute $X.a$ Once detecting this possible tradeoff gain, the algorithm considers the possibility of (1) moving $Y.d$ to $\Pi_{Y,1}$ or (2) moving $X.b, X.c$ to $\Pi_{X,2}$. There are, of course, global ramifications of moving an attribute from one partition to another. Any algorithm that does so must check **all** other productions containing an instance of Y or X to make sure that moving the attributes does not create a new spanning attribute or cause a circularity in some *DP* graph. It is probably best for any such algorithm to simply give up when a new spanning attribute is created, rather than then try to move the offending attribute in such a way as to keep it from being multi-visit also. Searching past the immediate context of a production $p : Y ::= X$



$$v: \cdots \rightarrow X.a, \rightarrow X.b, \rightarrow X.c, \downarrow_1 X, \cdots, \uparrow_1, \cdots, \downarrow_2 X, \cdots, \rightarrow Y.d, \uparrow_2$$

instance where saving attributes at the partitioning level
sacrifices them at the production level
- Figure 3.16 -

symbol will give an algorithm unacceptably high complexity (all symbols and all productions must be checked for each symbol processed).

3.8. Conclusions

This chapter gave a three stage algorithm for producing fewer multi-visit attributes for a simple multi-visit AG. The algorithm uses maximum network flow algorithms extensively. Combined, its three stages create: (1) partitions of attributes from *IDS* graphs, (2) visit sequences from *DP* graphs, and (3) postprocessed visit sequences which fix some of the problems left behind by the previous steps. The development of this algorithm was extensive, but its implementation should not be unwieldy. Much previous work exists in the area of maximum network flows, and algorithms solve the problem as fast as $O(mn \log n)$, where m, n are the number of vertices (attributes) and edges (dependencies) of the problem [Tar83].

Previous work exists in the area of making better partitions [FaY86], but their optimizations are a trivial subset of the ones performed by the maximum network flow solutions given here.

Little previous work exists in the area of making better visit sequences from *DP* graphs. The heuristic proposed as a solution here is ''almost'' optimal (in reality results will depend on the order in which the *DP* graphs are analyzed).

The incorporation of these results (plus those of Chapter II) into a simple multi-visit based system should easily produce attribute evaluators whose storage requirements are much better (for non-trivial input) than existing compiler writing systems.

CHAPTER IV

STRUCTURE TREE EVALUATION DURING LR PARSING

4.1. Introduction

As pointed out in Chapter I, the ability to integrate attribute analysis during the parsing phase of compilation has significant space and runtime advantages. There is, correspondingly, a considerable body of research that has uncovered the "LR-attributed" AGs, which can be evaluated entirely without a structure tree, during an LR parse [JoM80] [Tar82] [SaI86]. However, the class of LR-attributed grammars seems expressively weak; to date, only subsets of languages like Pascal and PL/C have been successfully written within this AG class [SIN85]. In this thesis, we propose a new type of attribute evaluator which, like the LR-attributed method, also works simultaneously with an LR parser. This new type of evaluator allows the parser to periodically pause and call for attribute evaluation of the structure tree it has built "so far", after which the evaluated structure tree fragment can be "thrown away." In this way, the enormous peak storage requirements of traditional attribute evaluators are avoided, while the expressive power of the simple multi-visit AG class is retained.

In the following, we discuss structure tree evaluation during parsing in the context of "LR" parsing[8]. The theory behind this form of parsing is well understood and treatments of it in the literature abound. Rather than reinvent the

[8]The integration of parsing and structure tree evaluation during "LL" parsing is not dealt with explicitly in this thesis since LL grammars are a proper subset of the LR grammars. In that sense, the same principles apply for LL as for LR.

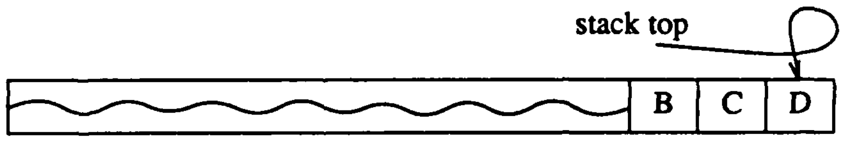
wheel here, we point the unfamiliar reader to the very excellent introductions of the subject in [WaG83], [AhU77], [HoU79].

Consider a typical state of an LR parser as it is depicted in Figure 4.1a. The parser has moved into a state where it is just about to "reduce" the parse stack by the production

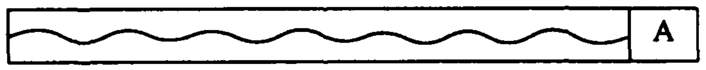
p: $A ::= B\ C\ D$

"Reduction" means that the parser will replace the top three nodes on the stack (containing B, C, D) by a single node containing A, as in Figure 4.1b. Parsing will thereafter continue as normal.

In the traditional model of attribute evaluation (in which the attribute evaluator is segregated from the parser) the parser has the responsibility for allocating nodes of the structure tree and attaching them to each other according to the productions it uses to recognize an input string. For example, Figure 4.2 shows what happens when a "structure tree building" LR parser is in the process of



(a) - parse stack just before reduction

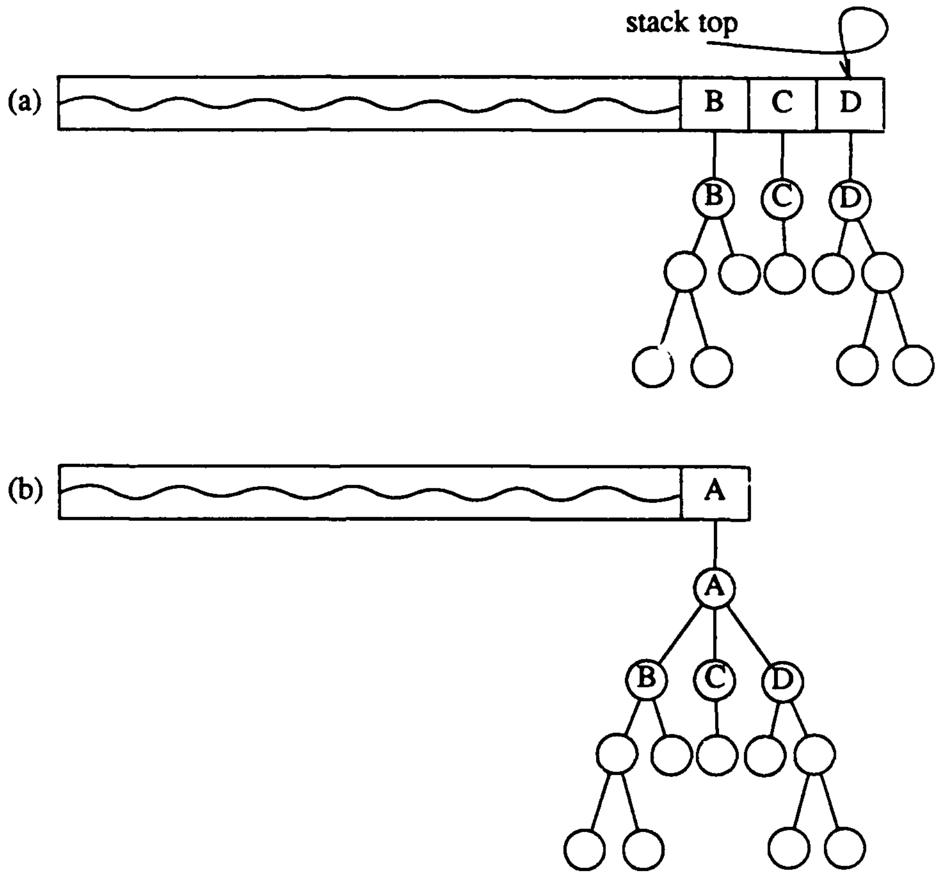


(b) - parse stack just after reduction

Parse stack before and after reduction by the production
$A ::= B\ C\ D$

- Figure 4.1 -

recognizing the same production p as in Figure 4.1. At the point where the parser is just about reduce by the production p to the symbol A, we can assume that it has already created the subtrees having B, C and D as root symbols, as in Figure 4.2a. At the time of reduction, the parser executes an *ATTACH* action which allocates a new node for the symbol A, and attaches the trees for B, C and D as A's children, as depicted in Figure 4.2b. Parsing thereafter continues as normal, and eventually builds the structure tree for the entire source input in this fashion.



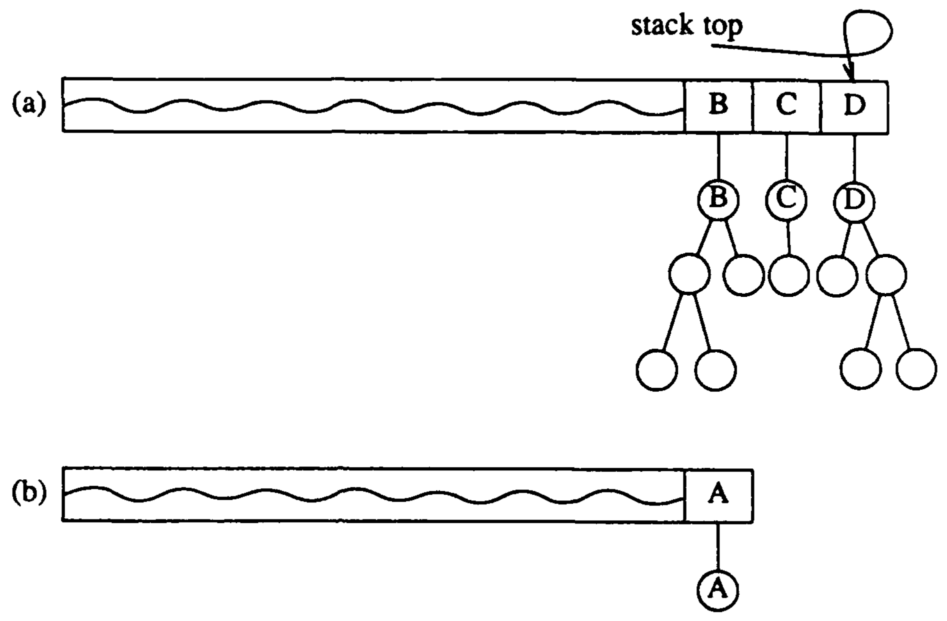
Parse stack before and after reduction by the production
$A ::= B\ C\ D$ by a structure tree building parser

- Figure 4.2 -

In the proposed model of an "integrated" parser and attribute evaluator, when the parser is in the process of reducing by certain productions, an *EVAL* action is called to perform attribute evaluation on the structure tree built "so far." Figure 4.3 shows what can happen for an integrated evaluator when the parser is just about to reduce the parse stack by the production p given above. If the inherited attributes of A were somehow available at the point in the parse shown in Figure 4.3a, then a tree walk evaluator (TWA, as described in Chapter I) could be used to evaluate that subtree with parent node A **immediately** after the *ATTACH* action makes B, C, and D subtrees of A. After this structure tree fragment has been evaluated, only the root node is required for the evaluation of the rest of the structure tree, so we can "collapse" the tree, deallocating all but the A node. Figure 4.3b shows the parse stack configuration that results after integrated evaluation and collapsing. We can easily see that the storage requirement at this point of the parse is much smaller than at the same point for Figure 4.2b.

4.2. Parse time structure tree evaluation

For our purposes, it suffices to point out that the decision algorithms in [JoM80] [SaI86] determine, for a particular inherited attribute $X.a$, whether or not that attribute will be computable when the parser reduces by some production having X as the parent symbol. It is clear that this is exactly the information we need to determine when to stop the parser and evaluate a structure tree fragment: if the decision algorithm says that all $X.a \in AI(X)$ are available (in the parse stack) at the point where a production is reduced to X, then we say X is "parse time tree evaluable" (PTTE). Correspondingly, a *TREE_EVAL* action can be called during reduction of any production having X as parent symbol. In Figure 4.3, assuming the symbols B, C, and D are not PTTE, their structure tree fragments are not evaluated



Parse stack before and after reduction by the production $A ::= B\ C\ D$ using an integrated parser/attribute evaluator

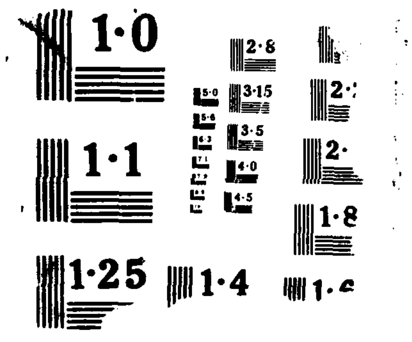
- Figure 4.3 -

and collapsed during earlier parse stack reductions, but instead are carried along with the parse until the PTTE symbol A is found.

A TWA used for integrated attribute evaluation cannot be the same as one used for segregated evaluation in general. Suppose two TWAs, TWA^S and TWA^I are built for a single AG containing the production p of Figure 4.3, using the segregated and integrated methods of attribute evaluation, respectively. In TWA^S, the visit sequence for any production q having A as a child symbol would perform at least one action of the form $\downarrow_i A$. However, it is clear that in TWA^I, once a subtree has been created for which A is a child symbol, all the attributes (and subtree) of A have already been evaluated during an earlier parsing step, and so having the action $\downarrow_i A$ in the visit sequence for q is erroneous. Instead, during runtime for

AD-A195 402 THE OPTIMIZATION OF AUTOMATICALLY GENERATED COMPILERS 2/2
(U) COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE
M L HALL 1987 N00014-86-K-0204

UNCLASSIFIED F/G 12/5 NL


1·0
1·1
1·25
1·4
2·8
3·15
3·5
4·0
4·5
5·0
5·6
1·8

TWA^{I}, A should be treated as a "magic terminal" whose attributes have already been evaluated, and should not be visited.

4.2.1. Creation of visit sequences

To see how to build the visit sequences for the productions of an AG when using an integrated evaluator, consider the following table:

	(1)	(2)	(3)	(4)
Parent Symbol	PTTE	PTTE	M-VIS	M-VIS
Child Symbols	PTTE	M-VIS	PTTE	M-VIS

This table describes the possible characteristics of the parent and child nodes of a production. "M-VIS" means that at least one inherited attribute of the symbol is not available upon reduction by that symbol (ie, the symbol is not PTTE). If the entry M-VIS occurs in any "child" slot of the table, that means at least one child is M-VIS. If the PTTE entry occurs in any child slot of the table, that means all child symbols of that production are PTTE.

Suppose we have a production p matching any of the situations (1) through (4) of the table, where p has the form:

p: $A ::= B\ C\ D$

To create a visit sequence for this production, the partitions of the symbols and the graph $DP(p)$ is computed in the usual way. However, any PTTE child symbol will be treated as a "magic terminal" by the integrated TWA, and so its contributions to the dependencies in $DP(p)$ should be deleted before creation of the visit sequence.

4.2.2. Insertion of TREE_EVAL and ATTACH actions

We assume the existence of a parser generator system like YACC [Joh75] which allows the insertion of user-supplied code to be evaluated during a reduction by the parser. An example of a typical production in a YACC specification is:

$$A ::= B\ C\ D \quad \{user-supplied\ \ code\}$$

This specifies that the user-supplied code is to be executed in a running parser whenever this production is recognized (ie, at reduction time). This is exactly the functionality required to get the parser to pause and call the *ATTACH* and *TREE_EVAL* procedures in the proposed integrated evaluation method.

When inserting the user supplied code for any production matching any of situations (1) through (4) of the above table, we must insert an *ATTACH* call to build the structure tree as before.

When inserting the user supplied code for any production p matching situations (1) or (2) of the above table, where p is of the form:

$$A ::= B\ C\ D$$

a *TREE_EVAL* call should be inserted since A is PTTE (recall that *TREE_EVAL* applies the TWA and deallocates the subtree nodes). For p matching situations (3) and (4), A is not PTTE, and so only the *ATTACH* call is made; evaluation will have to come later.

The previous description of a PTTE evaluator has actually been simplified for illustrative purposes. The following proposes some additional features for integrated attribute evaluators which makes their performance even better.

4.3. Optimizations

(1) In the above implementation of an attribute evaluator, no advantage was taken of symbols for which **all** attributes can be evaluated at parse time (using the LR-attributed method). We call these symbols "parse time evaluable" (the abbreviation for this is PTE, which conspicuously differs from PTTE because the "T" which stands for "tree" is missing). Experience using LR-attributed grammars has shown that a significant number of common programming language constructs can be evaluated using the LR-attributed method [SIN85]. We therefore expect that sizable chunks of a typical AG will require no structure tree to evaluate it. For productions where all symbols are PTE, we can forego the insertion of *TREE_EVAL* and *ATTACH* actions into the parsing grammar and allow the LR-attributed method to evaluate and store all attributes in the parse stack. When a PTE symbol shows up as a child of some non-PTE production, then we treat it as a "magic terminal" just as for PTTE child symbols. This additional feature should further reduce space requirements and also provide a modest runtime speedup.

(2) In the event that a symbol is not PTTE, there might still be **some** of its attributes that are available at parse time. In this case, the evaluation method described above computes these attributes **twice:** once when the attribute is being propagated along the parse stack using the LR-attributed method, and then again when the structure tree node is being evaluated by the TWA. Suppose one such attribute is $X.b$. We can avoid recomputation by copying its value from the parse stack to the structure tree node for X when the parser reduces a production to X. When this optimization is included, the action $\rightarrow X.b$ should be removed from all visit sequences (so that the TWA does not recompute $X.b$ during *TREE_EVAL*).

(3) The previous discussion assumes that all attributes in a structure tree fragment are stored in the tree nodes. We can implement attributes as global stacks and variables as discussed in Chapters I-III by considering each PTTE symbol as the root of a complete AG and analyzing the grammar for storage optimizations as before. However, if it turns out that an attribute cannot be implemented using a global stack for **one** of the new AG fragments, then it cannot be implemented as a global stack for **any** of them. Suppose a production

$$p: A ::= B\ C\ D$$

is used in two AG fragments, one where $B.a$ can be implemented using a stack and one where it must be in the structure tree. In order to avoid having two visit sequences for p, which are used depending on the context p occurs in, we choose to have a single visit sequence where $B.a$ is stored in the structure tree.

(4) Consider the AG fragment:

$$\begin{array}{l} A ::= B\ C\ D \\ \qquad r_1: B.b := f(C.c); \\ \qquad r_2: D.d := g(B.b); \\ \qquad \qquad \vdots \end{array}$$

The attribute $B.b$ is not PTTE since it depends on the value of $C.c$ (this dependence constitutes the propagation of information from right to left, which is strictly forbidden in LR-attributed grammars). Consequently, the attribute $D.d$ is not PTTE since it depends on $B.b$. Suppose A is PTTE, so the parsing actions for this production look like:

$$A ::= B\ C\ D \quad \{ATTACH, TREE\_EVAL\}$$

Let us further suppose that $B.b$ is the only inherited attribute of B which is not PTTE, and that $D.d$ is the only inherited attribute of D which is not PTTE. Consider the following modification of the above parsing grammar fragment:

$A ::= B\ C\ \langle act \rangle\ D$
$\quad \{ATTACH, TREE\_EVAL\}$
$\langle act \rangle ::=$
$\quad \{TREE\_EVAL(top\_of\_stack), TREE\_EVAL(top\_of\_stack - 1)\}$

The insertion of the symbol $\langle act \rangle$ serves simply to evaluate the trees for C and B before the parser accepts **any** of the string corresponding to D. Symbols like $\langle act \rangle$ are called "null-nonterminals" because they recognize the empty string. Recall that $D.d$ was previously not PTTE because of its dependence on the non-PTTE attribute $B.b$. However, with the insertion of $\langle act \rangle$ into the grammar, the value of $B.b$ is available before any of the subtree for D is built, which means that $D.d$ is now parse time evaluable according to the LR-attributed method! (this method of null-nonterminal insertion actually **increases** the number of attributes evaluable using the LR-attributed method, a surprising result). Visit sequences can now be modified to reflect the fact that D's attributes are evaluable without a structure tree, which results in a commensurate performance gain.

Unfortunately, we cannot insert null-nonterminals into productions in any position we would like, as this can destroy the LR property of the overall grammar [Joh75]. However, [PuB80] gives a set of algorithms for determining which positions in a production are safe to place null-nonterminals. We can use this information when trying to perform "early" structure tree evaluation, inserting null-nonterminals only in those places which are deemed safe by analysis[9].

[9]The use of LL grammars in this case simplifies matters considerably. A null-nonterminal can be inserted anywhere in any LL production; all positions in an LL grammar are "free" [PuB80].

4.4. Discussion

Hand-analysis of a complete Pascal AG (modified from the AG in [KHZ82] to do a better job of propagating "environment" attributes) shows that at least the symbol *BLOCK* (as in *BLOCK* ::= *DECLARATIONS STATEMENTS*) is PTTE. This means that the peak storage required to process a Pascal program will get no larger than the structure tree storage required to evaluate the largest procedure or function using this evaluation method. This is vastly different than the structure tree storage required to traditionally evaluate a Pascal program, which grows according to the length of the **entire** input.

We expect that a working compiler generator system based on integrated evaluation, having reasonable performance diagnostics, would allow us to iteratively modify this Pascal AG until it contained even better evaluation properties, such as building and evaluating trees only for statements (where good code must be generated). In this case, the information supplied by the generator system would be instrumental in figuring out how to make the AG have better integration characteristics.

The only cost increase associated with performing integrated evaluation (in the optimized form) seems to be the overhead in calling the *TREE_EVAL* routine. *TREE_EVAL* is called only once in the segregated method, whereas it can be called any number of times in the integrated method. Of course, even in the worst case, the same amount of "overall work" is performed by *TREE_EVAL* for either method. To justify this overhead cost, we point out that a compiler generated from a Pascal subset AG using the LR-attributed method runs within 12% of the time required by a hand written compiler for the same subset [SIN85]. This is a significant speed up over the runtime of an attribute evaluator which builds the entire structure tree [Gra85]. Since our integrated evaluation method produces even more LR-attributed attributes than the strict LR-attributed evaluation method (as shown in optimization (4) above), we expect to benefit from this speed up also. Even in the worst case, the integrated method performs exactly as the segregated method does, and so we do not expect to suffer worse results than the segregated method in any situation.

CHAPTER V

CONCLUSIONS

4.5. Overview

This thesis provided algorithms to extend the class of attributes that can be implemented using global stacks and variables. This extension resulted from a better analysis of visit sequences to determine when certain attributes are stackable. Taking advantage of this, an algorithm was later produced which actually creates visit sequences for which even more attributes can be stored using global stacks (and variables). In addition to these attribute storage improvements, a method was described whereby parsing and structure tree evaluation could be integrated. It is clear from the development of these ideas that their implementation could significantly reduce the storage requirements of compilers generated from attribute grammars. An implementation effort to investigate this claim is forthcoming.

5.1. Is parse time evaluation sufficient?

On first inspection, it may appear that the results of Chapter IV obviate the need for the storage optimizations given in Chapters II and III. Since it looks like the sizes of structure tree fragments built and evaluated using an integrated strategy will always be small, is it really necessary to store attributes in stacks and variables? The answer to this question is obvious when one considers that the Pascal AG as described in [KHZ82] has only **one** PTTE symbol: *program*. This means that, for this particular AG, the only time a parser can stop and call an attribute evaluator is when it has placed the *program* symbol on the parsing stack. But placement of this symbol on the stack implies that all of the input has been parsed, and so the structure tree for the **entire** input has to be passed to the attribute evaluator. If no attribute storage optimizations are performed for this AG, then the PTTE strategy (in this case) is no better than the naive approach to structure tree implementation mentioned in Chapter I (which requires gigantic amounts of storage).

Using a different argument, it is useful to allow a compiler writing system to accept AGs for which it will not be able to produce an integrated evaluator at first. Language designers like to experiment with and evaluate new language features before expending the effort required to build a high-performance compiler. Any reasonable compiler for their language would satisfy their needs during development, and this is certainly achievable using global stacks and variables. In this sense, the use of stacks and variables in addition to PTTE provides a two-level approach to language design, whereby designers first get a correct and reasonably efficient implementation of a compiler for their language, and later on can concentrate on "optimizing" the compiler. An especially nice feature of this two-level approach is that the optimization is incorporated piecemeal at the **specification** level, and a working compiler exists all the while. Contrast this to an LR-attributed system, where no evaluator can exist for an AG until **all** its symbols are PTE.

As a final argument, we feel it is unreasonable to expect that LR-attributed AGs will **ever** exist for complicated languages like Ada. We likewise do not expect that there will ever be an Ada AG which contains a huge number of PTTE symbols. Only the use of global stacks and variables will ensure that reasonable evaluators can be created for these AGs.

Each of these arguments is an important reason to incorporate the implementation of attributes using global stacks and variables in an automatically generated compiler.

5.2. Significance of work

Of the 150 or so publications in the field of attribute grammars cited in [Rai80], some 30 are concerned solely with the reduction of storage for attributes. Moreover, the most recent publication cited in this list is dated 1979 (there are many more papers on the topic that have come out since then). As of this writing, the most recent paper on the storage problem for AGs is [FaY86], which excellently argues that the job still has not been solved satisfactorily. Their paper calls for the following improvements to AG based systems:

(1) Better analysis to implement more attributes as global variables.

(2) Better analysis to implement more attributes as global stacks.

(3) Better analysis to implement more stackable attributes using "push from above and pop from below" (or vice-versa) where possible.

(4) The creation of "smart" visit sequences in the effort to more ably solve the three problems above.

This thesis significantly improves the present methods for attribute storage optimization in each of these four areas. Therefore any existing compiler generation system based on AGs can benefit by incorporating the theory and algorithms developed herein:

(1) The *CAN_EVAL* matrix of Chapter II allows any existing system that implements attributes as global variables to do a more thorough analysis.

(2) The extended definition of "single visit" in Chapter II and the algorithms given to stack attributes can be used to implement many more attributes as

global stacks than any current system provides. Of the twenty attributes left in the tree for the Pascal AG in [KHZ82], inspection shows that more than 7 are global stacks under the new definition (without even considering the increase which might occur with better partitioning). This number may seem small, but removing even a single attribute from the structure tree can sometimes result in significant space savings. Consider the production

identifier ::= *name*

of the Pascal AG. Supposing that a *name* will occur 2-3 times per input line of Pascal, for a 10,000 line program, 20,000-30,000 instances of this production will occur in a struc.ure tree. If one 4-byte attribute is removed from the *identifier* symbol, this results in a 80,000-120,000 byte storage reduction!

(3) The class of attributes in this work which are stackable "from below", namely all synthesized single visit attributes and all inherited single visit attributes in BnNF, is much larger than proposed or implemented elsewhere in the current literature. As pointed out in [FaY86], this will produce better stack depth and runtime statistics than systems which use purely "push and pop from above" stacking.

(4) The method of producing visit sequences given here is easily the most sophisticated of those in the current literature. Its performance in practice has yet to be determined, of course, but it is clear from the arguments of Chapter III that this method will allow better storage than either of the OAG or AAG methods (for AGs of sufficient complexity to take advantage of the power of the "near as possible" *NAP* () partitioning procedure).

In addition to these improvements, the scheme proposed here of integrating structure tree evaluation and parsing is completely new to the literature, and its potential for reducing peak structure tree storage requirements is very clear. Moreover, we think that the notion of a PTTE symbol, which is indeed quite close to the notion of a "single visit" symbol, is an easy one for compiler writers to grasp. We therefore expect AG writers to effectively go through the process of "optimization by specification" with the goal of making more attributes PTTE, iteratively refining their AGs to produce compilers which run faster and need less storage.

5.3. Future directions

The optimizations proposed in this thesis seem promising to reduce the storage requirements of compilers, but it is clear that they will not **significantly** increase compiler speeds. The experiments and measurements in [Gra85] [Wai86a] lead to the conclusion that drastic speedups in the runtime of generated compilers are not going to take place unless existing AG specification languages are changed. The strictly applicative nature of these languages, useful for formal analysis, makes it very difficult to generate extremely fast compilers. Promising future research points to the inclusion of abstract data types (ADTs) [GuH78] [LiZ75] into AG specification languages, which will make it possible to continue formally analyzing AGs, while permitting more efficient implementations.

A third reason why compiler writers do not use AGs (besides space and speed problems) is that they are difficult to use and apply to some well-understood programming language constructs. This was a complaint often voiced by the AG writers for the GMD Modula-2 compiler project [Wai85b]. We have initiated an investigation of this problem, and propose to experiment with several new language constructs (control and data) which may make the process of writing (and reading) attribute grammars easier.

REFERENCES

[AhU77] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading Ma., 1977.

[Boc76] G. V. Bochmann, "Semantics Evaluated From Left To Right", *Comm. of the ACM 19*, 2 (February 1976), 55-62.

[DeK75] F. L. DeRemer and H. Kron, "Programming-in-the-large versus Programming-in-the-small", in *Proc. of the International Conf. on Reliable Software*, 1975, 114-121.

[EnF82] J. Englefriet and G. File, "Simple Multi-Visit Attribute Grammars", *J. Computer and System Sciences 24* (1982), 283-314.

[Fan72] I. Fang, "FOLDS: a declarative formal language definition system", STAN-CS-72-329, C.S. Dept, Stanford University, Stanford Ca., December 1972.

[Far82] R. Farrow, "Linguist-86 Yet another translator writing system based on attribute grammars", *Proceedings of the ACM SIGPLAN Notices 82 Symposium on Compiler Construction*, June 1982.

[FaY86] R. Farrow and D. Yellin, "A Comparison Of Storage Optimizations in Automatically-Generated Attribute Evaluators", *Acta Inf. 23*, 5 (1986), 393-427.

[Gal81] M. Gallucci, "SAM/SAL An Experiment Using an Attributed Grammar", Ph.D. Thesis, Dept. of Computer Science, University of Colorado, Boulder, 1981.

[GaJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Co., San Francisco, 1979.

[Gra85] B. Gray, "Comparing Semantic Analysis Efficiency of a GAG Generated Compiler vs Hand Written Compilers", ECE690 Final Project, Univ. of Colorado, Dec. 1985.

[GuH78] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types", *Acta Inf. 10* (1978), 27-52.

[HoU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[JaP75] M. Jazayeri and D. Pozefsky, "Alternating Semantic Evaluator", *Proceedings of the ACM Annual Conference*, October 1975, 230-234.

[JaP81] M. Jazayeri and D. Pozefsky, "Space-Efficient Storage Management in an Attribute Evaluator", *Trans. Prog. Lang and Systems 3*, 4 (October 1981), 388-404.

[Joh75] S. C. Johnson, "YACC-yet another compiler compiler", CSTR-32, Bell Laboratories, Murray Hill, N.J., 1975.

[Joh80] S. C. Johnson, "A Tour Through the Portable C Compiler", in *UNIX Documentation*, Bell Telephone Laboratories, Murray Hill, NJ, 1980.

[JoM80] N. D. Jones and C. M. Madsen, "Attribute-Influenced LR Parsing", in *Proc of Aarhus Workshop in Semantics Driven Compiler Generation*, N. D. Jones (editor), Springer Verlag, 1980, 393-407.

[Kam83] T. Kamimura, "Tree Automata and Attribute Grammars", *Information and Control 57* (1983), 1-20.

[Kas80] U. Kastens, "Ordered Attribute Grammars", *Acta Inf. 13* (1980), 229-256.

[KHZ82] U. Kastens, B. Hutt and E. Zimmermann, *GAG: A Practical Compiler Generator*, Springer Verlag, 1982.

[Kas86] U. Kastens, Personal communications, 1986.

[Knu68] D. E. Knuth, "Semantics of Context-Free Languages", *Mathematical Systems Theory 2*, 2 (1968), 127-144.

[Knu71] D. E. Knuth, "Semantics of Context-free Languages: Correction", *Mathematical Systems Theory 5* (Mar. 1971), 95-96.

[Kos84] K. Koskimies, "A Specification Language for One-Pass Semantic Analysis", *SIGPLAN Notices 19*, 6 (June 1984), 179-189.

[LiZ75] B. H. Liskov and S. N. Zilles, "Specification Techniques for Data Abstractions", *IEEE Transactions on Software Engineering SE-1*, 1 (March 1975), 7-19.

[PuB80] P. Purdom and C. Brown, "Semantic Routines and LR(k) Parsers", *Acta Inf. 14*, 1 (1980), 299-315.

[Rai80] K. Raiha, Bibliography on Attribute Grammars, Unpublished Report, Dept. of Computer Science, Univ. of Helsinki., 1980.

[Saa78] M. Saarinen, "On constructing efficient evaluators for Attribute Grammars", in *Automata, Languages, and Programming: 5th Colloquium*, Ausiello and C. Bohm (editor), Springer-Verlag, 1978.

[SIN85] M. Sassa, H. Ishizuka and I. Nakata, "A Compiler Generator Based on LR-attributed Grammars", Tech. Memo PL-7, Institute of Information Sciences and Electronics, University of Tsukuba: Sakura-Mura, Niiari-Gun, Ibaraki-Ken 305, Japan, Nov. 1985.

[SaI86] M. Sassa and H. Ishizuka, "A contribution to LR-attributed grammars", *J. Inf. Proc (to appear) 8*, 3 (1986).

[ScJ83] P. Schulthess and C. Jacobi, ''Anatomy of a Small Pascal Compiler'', *IEEE Trans. on Software Eng. SE-9* (1983), 185-191.

[Tar82] J. Tarhio, ''Attribute Evaluation During LR Parsing'', Report A-1982-4, Univ. Of Helsinki, Finland, 1982.

[Tar83] R. A. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Pa, 1983.

[WaG83] W. M. Waite and G. Goos, *Compiler Construction*, Springer-Verlag, 1983.

[Wai85a] W. M. Waite, ''The Cost of a Generated Parser'', *Software—Practice & Experience 15*, 3 (March 1985), 221-237.

[Wai85b] W. M. Waite, Personal communications concerning the GMD Modula-2 compiler project, 1985.

[Wai86a] W. M. Waite, GAG/ADT, Unpublished Report, Dept. of Electrical and Computer Engineering, Univ. of Colorado., 1986.

[Wai86b] W. M. Waite, ''The Cost Of Lexical Analysis'', *Software—Practice & Experience 16*, 5 (May 1986), 473-488.